

The Official
GEOSTM
Programmer's Reference Guide

Berkeley SoftworksTM
Michael Farr

THE OFFICIAL GEOS PROGRAMMERS REFERENCE GUIDE

GEOS™ is a trademark of Berkeley Softworks
Commodore 64, Commodore 64c, and Commodore 128
are trademarks of Commodore Business Machines, Inc.

All rights reserved
Copyright © 1987 Berkeley Softworks

Reproduced by Creative Micro Designs, Inc. with permission
by Geoworks (formerly Berkeley Softworks).

Throughout this guide, the trade names and trademarks of
some companies and products have been used, and no
such uses are intended to convey endorsement of or
other affiliations with this guide.

This book may not be reproduced in whole or in part, by mechanical,
electrical, electronic, or any other means without the express
written permission of the copyright owner.

Creative Micro Designs, Inc.
P.O. Box 646, 15 Benton Drive
East Longmeadow, MA 01028-0646
(413) 525-0023

Corrections and Addendum to The Official GEOS Programmers Reference Guide

****PRGTOGEOS Conversion Program (chapter 3) Corrections****

The PRGTOGEOS utility described in chapter 3 does not work. With the following corrections, you should have no problem with it.

Page 48 Corrections

Replace the "psect 400" with:

```
.psect $304      ;The fileheader does not actually start at $304
                  ;This psect is necessary so that the fileheader
                  ;occupies the first 252 bytes of the PRG file
                  ;which your assembler creates.
```

The comments below "FileHeader!" should read:

```
;The first four bytes of the fileheader will be written to the
;header block by the PRGTOGEOS basic program during conversion.
;For your information, the first four bytes will contain:
;.byte $00,$ff    ;null pointer to next block
;.byte 3,21       ;icon width and height
```

Then these comments follow:

```
;The rest of the header will end up in bytes $04-$ff of the first
;block of the PRG file. When this file is converted, this first
;block will become the program's header block, and the header
;information from $04 to $ff in the block will already be in the
;correct place.
```

The rest of the header from P48 follows from here, starting with the .byte (\$80+63) line and ending with the .byte 0,0,0,0 line. Then replace everything on the page below that line with the following:

```
;the assembler should fill any free space in here with $00's,
;until the location counter equals $400.
```

```
.psect $400      ;The code following this psect will be written
                  ;into block 2 of the PRG file.
```

InitCode:

```
lda #21      ;here are two sample  
sta r0L      ;lines. Change these.
```

```
;rest of application code goes here
```

EndCode:

```
;This label is use by the header block to store  
;the ending address of the application code.
```

THIS CONCLUDES the changes to page 48. The only other necessary changes are in the PRGTOGEOS basic program.

Pages 50-52 Corrections

Now that we have corrected the header block on page 48, we need to change several lines of the PRGTOGEOS basic program which is listed on pages 50-52. Replace the following lines with the lines listed below:

```
80 INPUT "HOUR      (EX: 14) ";H
```

```
180 T$=HT$:S$=HS$:GOSUB 1000
```

```
210 FOR I=2 TO 68
```

```
310 FOR I=0 TO (32*E)+2
```

After you make the above corrections, do the following:

- DELETE line 240 (240 GET#2,CT\$)
- Make sure that line 3050 reads: 3050 FOR I=I TO 31

That should do it. Good luck!

****p222 corections****

The DBGETFILES equate described on P222 in the GEOS REF MAN should have a value of 16, not 15 as it appears in the manual. See P405 for confirmation. Thanks to RickR9 for the tip. :)

****Macros****

The Reference Manual does not have a page describing the macros we use in the source listings. Here are definitions for the 5 most commonly used:

LoadB

LoadB is used to stuff a constant value into a single-byte memory location.

```
.macro LoadB addr,value  
    lda    #value  
    sta    addr .endm
```

sample usages:

```
LoadB r0H,#52  
LoadB $0001,$$35  
LoadB mouseYPosition,#34
```

LoadW

LoadW is used to stuff a word constant into a word (two sequential bytes) memory location.

```
.macro LoadW addr,value  
    lda    #[value  
    sta    addr  
    lda    #]value  
    sta    addr+1 .endm
```

Note that the [char is used to get the low byte value of a word constant, and] is used to get the high byte value...

Sample usage:

```
LoadW r0,$$4000
```

MoveB

MoveB is used to copy data from one memory location to another.

```
.macro MoveB source,dest  
    lda    source  
    sta    dest .endm
```

Sample usage:

```
MoveB r0H,r2H ;set r2H=r0H
```

MoveW

Same as MoveB, but copies a word.

```
.macro MoveW source, dest
    lda    source
    sta    dest
    lda    source+1
    sta    dest+1 .endm
```

bra

This macro is used for unconditional relative branches.

```
.macro bra dest
    clv
    bvc dest .endm
```

Sample usage:

```
10$    nop
        nop
        bra 10$    ;loop indefinitely
```

****desk accesories in applications****

This note is for people writing applications which allow the user to run desk accessories.

Most GEOS applications that have been written are VLIR module swapping applications, and so the disk containing the application must always be in one of the active drives. The convention for placement of desk accessories is that they must be on the application disk. This way, the DAs which appear in the application's GEOS menu are always accessible. Applications of this type which allow the user to load data from the other drive must keep track of which drive is the application drive. For example, assume the application disk is in drive A and the data disk is in drive B. When the application is run from the deskTop, one of the first things it does is to read the disk's directory, grabbing the names of the DAs on that disk and stuffing them into the GEOS menu definition table. Then when the user has opened a datafile which is on the disk in drive B, and he selects a DA from the menu, the application must call OpenDisk to open drive A to run the DA. When the DA finishes, the application must call OpenDisk to re-open the data disk in drive B. If the system only has one drive, then there is no problem because the disk in drive A contains the application, the DAs, and the datafile; no drive switching is required when running DAs.

For applications which run entirely resident in memory (never swapping modules) you might be tempted to allow the user to remove the disk on a one-drive system and insert a different data disk. Applications such as the Icon Editor allow this. But be sure that if you allow this capability, that when the DA is selected AND the application disk has been removed, you must either 1) not allow the DA to run, or 2) put up a dialog box requesting that the application disk be inserted so that the DA can run.

****Appendix C (p432) Corrections****

DoneWithIO should read \$c25f

CopyString should read \$c265

****geopaint datafile structure****

The VLIR application data file that GeoPaint creates contains bitmap information for a page with dimensions of 640 horizontal by 720 vertical pixels.

This page is divided into 8 line high card-rows, just like the C-64 graphics screen. Each card-row is 640 pixels wide by 8 lines high, and requires 640 bytes of storage to describe the bitmap in that area. Also, each card-row requires 80 bytes of color information, one byte per card.

GeoPaint stores two card-rows worth of bitmap and color information in each VLIR record, in the following order:

```
640 bytes bitmap info for row #0
    (80 cards * 8 bytes each)
640 bytes bitmap info for row #1
80 bytes color info for row #0
    (80 cards * 1 byte each)
80 bytes color info for row #1
```

Note that the 640 bytes of bitmap information are structured by cards; the first eight bytes specify card #0, the next eight specify card #1, and so on. The first byte of each eight-byte group defines the top line of the card, the second byte defines the second line, and so forth. Within each byte, the most significant bit specifies the left-most pixel in an eight-pixel line within the card.

Each byte which defines color for a card is split up into two four-bit nibbles. The higher-order nibble (bits 7 through 4) specifies the color of the foreground in that card while the lower order nibble (bits 3 through 0) specifies the background color for that card.

Before these 1440 bytes are written out to the record, they are compacted using a run-length encoding scheme. This compaction technique is the same one used for bitmap images by GEOS. See page 89 of the Programmer's Reference Manual for an explanation. :)

128 GEOS Technical Reference Notes

December, 1987

This document provides some preliminary information about the differences between C64 GEOS and C128 GEOS. It is not intended to be comprehensive.

Compatibility with C64 GEOS Software

- Most C64 GEOS software will run under the C128 GEOS in 40 column mode.
- All data files, scraps, fonts, & printer drivers are identical under C64 and C128 GEOS.
- Input drivers are located at different addresses in the two machines, and hence are incompatible. We have added a new file type, INPUT_128, for C128 input drivers.
- As the deskTop is heavily tied into each OS, we've decided to give the 128 it's own desktop filename, "128 DESKTOP", so as to avoid confusion with the 64's "DESK TOP" file. (The deskTop is of file type "SYSTEM", and can't be renamed by the user).

128 Flags for Applications & Desk Accessories

In order for the 128 DESKTOP & other applications to know what files run in what mode, we've adopted a standard that should be used on ALL applications, desk accessories, & auto-execution programs. This flag is located in the header block of each of these programs. Since permanent file names are only 16 bytes long, we have left over 4 bytes that have been unused till now, but we've constantly been setting them to all 0's. The last of these bytes (see OFF_128_FLAGS) now has meaning to the 128 OS & DeskTop:

B7	B6	
0	0	runs under 128 GEOS, but in 40 column mode only
0	1	runs under 128 GEOS in 40 and 80 column modes
1	0	DOES NOT RUN under 128 GEOS (deskTop will display dialog box)
1	1	runs under 128 GEOS in 80 column mode only

Note: bits 5 through 0 are unused and should be 0. The 128 GEOS routines LdApplic and LdDeskAcc will return the error #INCOMPATIBLE if these flags in the header block do not allow running in the current mode.

Converting 64 GEOS software to run on the C128

First, you need to decide whether your software is simply going to be able to run in 40 col. mode, or whether it is to run in 40/80 column on the 128 only.

40 col. mode only on 128:

- 1) Chances are quite good your software already does.
- 2) If it doesn't, it's probably because you access BASIC – the 128 has a different BASIC, so you'll need to re-write that section of code to first see which OS you're running under, & then use the appropriate BASIC variables & jump entries.

40/80 col. on the C128 only:

- 1) Set the 128 flag as mentioned above in the save file to \$40
- 2) In 80 column mode, you'll need to widen your menus to accommodate the wider system font. We typically stuff these "right edge" values into the menu structure itself based on the current graphicsMode (\$80 is 80 column, \$00 is 40)
- 3) Most of the graphic changes you'll need to make can be accomplished by setting the high bit of every X position or width that you pass to the operating system. The 128 GEOS will ignore this bit if in 40 column mode, and double the value if in 80 column mode, thus automatically retaining the same sized image on the screen. Hence, 50+\$8000 is 50 pixels in 40 column mode and 100 pixels in 80 column mode.
- 4) If you're writing an application, add a "switch 40/80" option under the geos menu. The action for this should be to EOR graphicsMode with \$80, store it back, and call the routine SetNewMode. You'll then need to redraw the screen, now in the new graphics mode.
- 5) Nearly all x positions passed to the C128 GEOS can have the high bit set causing the position to automatically be doubled in 80 column mode. It has been noted that this always results in the low bit of the resulting word being a 0. This can make life difficult, if you desire to fill a pattern to the right edge of the screen, for instance. To solve this problem, I've modified the normalization routine: The 15th bit of the word continues to be the same "double me if 80 col." flag, but the 14th bit now has significance in 80 col. mode only – it becomes the low bit of the doubled word. So, if you want the right edge of the screen, use the value \$C000+319.

Sprites

The C64 contains a chip to handle sprites in hardware. Unfortunately, this chip is not available on the 128, so the functions of that chip have been simulated in software that is included in the 128 kernal. Most of the capabilities of the VIC chip have been taken care of, and if you are not doing exotic things with sprites your code may work with one or two changes.

The major changes include: sprite 0 (the cursor) is treated differently than any other sprite. The code for this beast has been optimized to get reasonably fast mouse response, with a resulting loss in functionality. You cannot double the cursor's size in either x or y. You cannot change the color of the cursor. The size of the cursor is limited to 16-pixels wide and 8 lines high. One added feature is the ability to add a white outline to the picture that is used for the cursor. This allows it to be seen while moving over a black background.

For the other 7 sprites, all the capabilities have been emulated except for color and collision detection. In addition, the 64th byte of the sprite picture definition (previously unused) is now used to provide some size info about the sprite. This is used to optimize the drawing code. Here are some problem areas to watch out for:

Writing directly to the screen:

Since the old sprite were handled with hardware, writing to the screen wasn't a problem. If you do it (system calls NOT included), then call "TempHideMouse" before the write. This will erase the cursor and any sprites you have enabled. You don't have to do anything to get them back, this is done automatically during the next main loop.

All sprite picture data:

All picture data should be adjusted to include the 64th byte. This byte has size information that is read by the software sprite routines, even if they are garbage values. The format of this byte is: high bit set means that the sprite is no more that 9 pixels wide (this means it can be shifted 7 times and still be contained in 2 bytes). The rest of the byte is a count of the scan lines in the sprite. You can either include this info as part of the picture definition, or stuff it into the right place with some special code.

Writing directly to the VIC chip:

This is generally ok, since the sprite emulation routines take the position and doubling info from the registers on the VIC chip, with the exception of the x position. The VIC chip allows 9 bits for x positions, which is not enough 640-wide screen. You should write the x position to the global variables "reqXpos0, reqXpos1..." (request x pos). These are full words, in consecutive locations. Better yet, use the "PosSprite" call in the kernel.

Reading values from the VIC chip:

This is also ok for the status values and for the y position. The x position can be useful also, if you use the PosSprite call. In addition to filling the global variables reqXpos0, etc., this call divides the x position by two and stuffs it into the VIC chip.

Using VIC chip collision detection:

This is iffy. The chip continues to operate, so if you are using the PosSprite call (see above) collisions should be detected with some loss of accuracy (the low bit). This has not been tested, so if you try it -- report the results here.

Writing to the VIC chip (or calling PosSprite, EnablSprite, DisablSprite) at interrupt level:

Don't do it. Since the mouse and the sprites are drawn at main loop, this causes subtle, irreproducible timing bugs that are impossible to get out.

Known bugs in release 1 of GEOS 128

- 1) If location \$1300 in application space is zero, then sprites in 80 column mode go haywire. All of our current applications that run in 80 column mode have put in a patch for this. Bug is in sprite code.
- 2) Doubling bitmaps through BitmapClip doesn't work.
- 3) i_BitmapClip needs call to TempHideMouse before being called.



128 GEOS Constants

December, 1987

```
;*****  
;This file contains additional constant definitions for applications which  
;will run under the GEOS 128 kernal. 12/11/87  
;*****
```

```
;The following equates define the numbers written to the "config"  
;register (location $FF00 in C-128). This register controls the memory map  
;of the C-128.
```

```
CIO_IN          = $7E ;60K RAM, 4K I/O space in  
CRAM_64K        = $7F ;64K RAM  
CKRNL_BAS_IO_IN = $40 ;kernal, I/O and basic ROM's mapped into memory  
CKRNL_IO_IN     = $4E ;Kernal ROM and I/O space mapped in
```

```
;Keyboard equates
```

```
KEY_HELP        = 25  
KEY_ALT         = 26  
KEY_ESC         = 27  
KEY_NOSCTRL     = 7  
KEY_ENTER       = 11
```

```
;128 screen size constants
```

```
SCREEN_BYTE_WIDTH = 80  
SCREEN_PIXEL_WIDTH = 640
```

```
;New GEOS file types:
```

```
INPUT_128        = 15 ;128 Input driver
```

```
;New # of file types, including NON_GEOS (=0)
```

```
NUM_FILE_TYPES   = 16
```

```
;The following equate can be used as an offset into a file's header block.  
;It points to the byte which contains flags which indicate if the program  
;will run under C-128 GEOS in 40 and/or 80 column modes. These flags are valid  
;for applications, desk accessories, and auto-exec files.
```

```
;
```

```
;B7 B6
```

```
;0 0 runs under 128 GEOS, but in 40 column mode only
```

```
;0 1 runs under 128 GEOS in 40 and 80 column modes
```

```
;1 0 DOES NOT RUN under 128 GEOS (deskTop will display dialog box)
```

```
;1 1 runs under 128 GEOS in 80 column mode only
```

```
OFF_128_FLAGS    = 96
```

```
;disk equates
DIR_1581_TRACK      = 40  ;track # reserved on 1581 disk for directory
DRV_1581             = 3   ;Drive type Commodore 1581
DRV_NETWORK          = 15  ;Drive type for GEOS geoNet "drive"

;equate for error value which indicates an attempt has been made to load
;an application which cannot be run under the current C128 graphics mode.
INCOMPATIBLE        = 14
```

128 GEOS Memory Map

December, 1987

```
;*****  
;This file contains additional memory map definitions for applications which  
;will run under the GEOS 128 kernal. 12/11/87.  
;*****
```

;Memory-management unit in C-128

```
mmu          = $D500  
VDC          = $D600
```

;Address of memory-map configuration register in C-128

```
config       = $FF00
```

;Misc addresses:

```
keyreg       = $d02f      ;C-128 keyboard register for # pad & other keys  
clkreg       = $d030      ;C-128 clock speed register
```

;Address of input driver

```
MOUSE_BASE   = $FD00  
END_MOUSE    = $FE80
```

;Note that the jump table entries for input driver routines have not moved;
;They are still at MOUSE_JMP (\$FE80). In 128 GEOS, there is one additional
;vector:

```
SetMouse     = MOUSE_JMP+9      ; ($FE89)
```

;Jump addresses within disk drivers -- these are only valid for non-1541
;disk drive types, and for the 128 version of the 1541 driver:

```
Get1stDirEntry = $9030      ;returns first dir entry  
GetNxtDirEntry = $9033      ;returns next dir entry  
AllocateBlock  = $9048      ;allocates specific block  
ReadLink      = $904B      ;like ReadBlock, but returns only 1st two bytes
```

;The following address holds info about the current 128 graphics mode:
;\$00 for VIC, \$80 for VDC 640*200, and \$C0 for VDC 640*400 (not yet
supported).

```
graphicsMode = $003F      ;holds current 128 graphics mode
```

;Misc vectors:

JumpIndX = \$9D80 ;address of routine used by 128 kernal

;This is a second GLOBAL memory area for GEOS. It is here so that these
;variables may be in the same place in V1.3 as they are running V1.2 with
;the V1.3 deskTop. This allows other input drivers to be auto-booted by
;those who have a V1.2 GEOS KERNAL, but have gotten the V1.3 deskTop via
;Q-LINK, or up-grade disk. They are EQUATED to be past the local GEOS
;variables, in fact at the end of the GEOS RAM space. Here they must
;permanently reside, for the sake of compatibility.

;Saved value of moby2 for context saving done in dlg boxes & desk accessories.
;Left out of original GEOS save code, put here so we don't screw up desk
;accessories, etc, that know the size of TOT_SRAM_SAVED above.

savedmoby2 = \$88bb

;copy of reg 24 in VDC for C128

screen80polarity = \$88bc

;Screen colors for 80 column mode on C128. Copy of reg 26 in VDC

screen80colors = \$88bd

;Holds current color mode for C128 color routines.

vdcClrMode = \$88be

;(4 bytes) 1 byte each reserved for disk drivers about each device
;(each driver may use differently)

driveData = \$88bf

;Number of 64K ram banks available in Ram Expansion Unit. 0 if none available.
ramExpSize = \$88c3

;If RAM expansion is in, Bank 0 is reserved for the kernal's use. This byte
;contains flags designating its usage:

;
; Bit 7: if 1, \$0000-\$78FF used by MoveData routine
; Bit 6: if 1, \$8300-\$B8FF holds disk drivers for drives A through C
; Bit 5: if 1, \$7900-\$7DFF is loaded with GEOS ram area \$8400-\$88FF by
; ToBasic routine when going to BASIC

```

; Bit 4:  if 1, $7E00-$82FF is loaded with reboot code by a setup AUTO-EXEC
;         file, which is loaded by the restart code in GEOS at $C000 if
;         this flag is set, at $6000, instead of loading GEOS_BOOT.
;         Also, in the area $B900-$FC3F is saved the kernal for fast
;         re-boot without system disk (depending on setup file).
;         This area should be updated when input devices are changed
;         (implemented in V1.3 deskTop)

```

```

sysRAMFlg      = $88c4

```

```

;This flag is changed from 0 to $FF after deskTop comes up for the first time
;after booting.

```

```

firstBoot      = $88c5

```

```

;(4 bytes) Current disk type for each drive (copied from diskType)

```

```

curType        = $88c6

```

```

;(4 bytes) RAM bank for each disk drive to use if drive type is RAM DISK
;or Shadowed Drive

```

```

ramBase        = $88c7

```

```

;(17 bytes) Holds name of current input device

```

```

inputDevName    = $88cb

```

```

;(18 bytes) Disk name of disk in drive C. (Padded with $a0)
DrCCurDkNm      = $88dc

```

```

;(18 bytes) Disk name of disk in drive D. (Padded with $a0)
DrDCurDkNm      = $88ee

```

```

;(256 bytes) 2nd directory header block, for larger disk capacity drives
;(such as 1571)
dir2Head        = $8900

```

```

;-----

```



128 GEOS Routines

December, 1987

```
;*****  
;This file contains additional jump-table entries for applications which  
;will run under the GEOS 128 kernal. 12/11/87.  
;*****
```

```
TempHideMouse      = $c2d7  
SetMousePicture    = $c2da  
SetNewMode         = $c2dd  
NormalizeX         = $c2e0  
MoveBData          = $c2e3  
SwapBData          = $c2e6  
VerifyBData        = $c2e9  
DoBOp              = $c2ec  
AccessCache        = $c2ef  
HideOnlyMouse      = $c2f2  
SetColorMode       = $c2f5  
ColorCard          = $c2f8  
ColorRectangle     = $c2fb
```



Contents

CHAPTER 1	BASIC GEOS	1
	Introduction; Speaking the Same Language; The Basics, Double Clicks through otherPressVector; Getting Started; Summary; The GEOS Kernal Structure; Calling GEOS Kernal Routines; Non-Event Code; Steps in Designing a GEOS Application; Hi-Resolution Bit- Mapped Mode; MemoryMap; GEOS Kernal Version Bytes; Bank Switching and Configuring; Assembler Directives; What's to Come	
CHAPTER 2	ICONS AND MENUS	23
	Icons; DoIcons; Menus; Menu Selections; DoMenu; ReDoMenu; DoPreviousMenu; GotoFirstMenu; Getting Up and Running; File Header Block	

CONTENTS

CHAPTER 3	GEOS FILE IN PRG FORMAT	45
	TestApplication; PRGTOGEOS; PRGTOGEOS; Getting It to Run; The File Quantum Test	
CHAPTER 4	DRAWING WITH GEOS	67
	Drawing with Patterns; Display Buffering; Drawing Lines; DrawPoint; TestPoint; HorizontalLine; VerticalLine; InvertLine; ImprintLine; RecoverLine; DrawLine, i_DrawLine; Drawing Filled Regions; SetPattern; Rectangle, i_Rectangle; FrameRectangle, i_FrameRectangle InvertRectangle; RecoverRectangle, i_RecoverRectangle; ImprintRectangle, i_ImprintRectangle; Bit-Mapped Graphics; The Compaction Formats; BitmapUp, i_BitmapUp; BitmapClip; BitOtherClip; GraphicsString, i_GraphicsString; GetScanLine	
CHAPTER 5	TEXT IN GEOS	103
	Simple String Manipulation; PutString; PutDecimal; String Input; GetString	
CHAPTER 6	CHARACTER LEVEL ROUTINES	113
	GetNextChar; InitTextPrompt; PromptOn; PromptOff; Putchar; GetRealSize; GetCharWidth; Style Control; Fonts; Using Fonts; LoadCharSet; UseSystemFont	
CHAPTER 7	INPUT DRIVER	135
	The Standard Driver; What an Input Driver Does; InitMouse; Acceleration, Velocity, and Nonstandard Variables; SlowMouse; SlowMouse; UpdateMouse; UpdateMouse; Mouse Variables for Input Driver; The Mouse as Seen by the Application; StartMouseMode; MouseOff; MouseUp; Additional Mouse Control; IsMseInRegion; Mouse Variables for Applications; Joystick	

CONTENTS

CHAPTER 8	SPRITE SUPPORT	171
	DrawSprite; PosSprite; EnablSprite; DisableSprite	
CHAPTER 9	PROCESS SUPPORT	177
	InitProcess; RestartProcess; BlockProcess, UnblockProcess; FreezeProcess, UnfreezeProcess; Sleep; EnableProcess	
CHAPTER 10	MATH LIBRARY	187
	DShiftLeft – Double Precision Shift Left; DShiftRight – Double Precision Shift Right; BBMult – Multiply Byte*Byte; BMult – Multiply Word*Byte; DMult – Double Precision Multiply; DDiv – Double Precision Divide; DSDiv – Signed Double Precision Divide; Dabs – Double Precision Absolute Value; Dnegate – Signed Double Precision Negate; DDec – Decrement an Unsigned Word; GetRandom	
CHAPTER 11	GENERAL LIBRARY ROUTINES	199
	CopyString; CopyFString – Copy Fixed Length String; CmpString; CmpFString – Compare Fixed Length String; Panic – Roll Over and Die; MoveDate, i_MoveDate; ClearRam; FillRam, i_FillRam; InitRam; CallRoutine; GetSerialNumber; ToBasic; FirstInit; CRC; ChangeDiskDevice	
CHAPTER 12	DIALOG BOXES	217
	DB Icons and Commands, DB Structure; Position Command; Dialog Box Commands; Dialog Box Example; openBox; Dialog Box Example: LoadBox/GetFiles; The Dialog Box Routines; DoDlgBox; RstrFrmDialog	

CONTENTS

CHAPTER 13	FILE SYSTEM	233
	The Foundation; Header Block; Using GEOS Disk Access Routines; Accessing the Serial Bus	
CHAPTER 14	HIGH-LEVEL FILE SYSTEM	251
	SetDevice; OpenDisk; GetPtrDkNm – Get Pointer to Current Disk Name; SetGEOSDisk; ChkDkGEOS; FindFTypes; GetFile; FindFile; SaveFile; DeleteFile; RenameFile; EnterDeskTop; CalcBlocksFree	
CHAPTER 15	INTERMEDIATE LEVEL	271
	FindFile; GetBlock; PutBlock; GetFHdrInfo; ReadFile; WriteFile; ReadByte; GetDirHead; PutDirHead; NewDisk; LdAppic – Load Application File; LdFile – Load File; GetFreeDirBlk – Get Free Directory Block; BlkAlloc; NxyBlkAlloc; SetNextFree; FindBAMBit; FreeBlock; SetGDirEntry; BldGDirEntry – Build GEOS Directory Entry; FollowChain; FastDelFile; FreeFile	
CHAPTER 16	PRIMITIVE ROUTINES	305
	InitForIO; DineWithIO; PurgeTurbo; EnterTurbo; ReadBlock; WriteBlock	
CHAPTER 17	VLIR FILES	313
	VLIR Routines; Error Messages; OpenRecordFile; CloseRecordFile; UpdateRecordFile; PreviousRecord, NextRecord, PointRecord; DeleteRecord; WriteRecord; ReadRecord	
CHAPTER 18	PRINTER DRIVERS	325
	Dot Matrix Printer Type; Talking to Printers; Parallel Interface Questions; GEOS Printer Drivers; The Interface – For Graphic Printing; ASCII	

CONTENTS

Printing; Calling a Driver from an Application; Using
a Printer Driver from an Application; InitForPrint;
GetDimensions; StartPrint; PrintBuffer; StopPrint;
StartASCII; PrintASCII; Sample Printer Driver;
8-Bit Printer Driver; Resident Top-Level Routines;
Resident Subroutines

CHAPTER 19	COMMODORE DRIVER	363
	Commodore Compatible Printer Driver Description; Resident Top-Level Routines; Resident Subroutines; Commodore Specific Routines; Warm Start Configuration	
APPENDIX A	CONSTANTS	391
APPENDIX B	GLOBAL GEOS VARIABLES	409
APPENDIX C	ROUTINES	429
APPENDIX D	DATA FILE FORMATS	435
INDEX		447

Q

Q

Q

1.

Basic GEOS

Introduction

Welcome to programming under GEOS. If you are already a Commodore 64 (c64) programmer you will find your transition to GEOS to be smooth. If you are new to programming the c64, you will find that you'll progress quickly because GEOS takes care of many of the difficult details of programming, and lets you concentrate on your design.

This reference guide assumes a knowledge of assembly language programming, and a general familiarity with the c64 computer. A good assembly language programming book on the 6502 chip and a copy of the Commodore 64 Programmer's Reference Guide are good references to have handy.

GEOS stands for Graphic Environment Operating System, and as its name implies, GEOS uses graphic elements to provide a simple user interface and operating syst-

em. The philosophy of GEOS is to handle in a simple way much of the dirty work an application might otherwise have to perform: the disk handling, the bit-mapped screen manipulation, the menus, the icons, the dialog boxes, and printer and input device support.

Programmers who take full advantage of the features GEOS has to offer should be able to cut development time significantly and increase the quality of their applications at the same time. Many of these features, such as proportionally spaced fonts, or a disk turbo, would not make sense for programmers to design into each application. With GEOS these features are provided. In the time it takes to write simple text routines one can be using proportionally spaced fonts, menus, icons, and dialog boxes to provide a sharp, intuitive, and general user interface.

Using GEOS's menus, window, and other graphic features makes applications look better, and easier to use. GEOS makes it easier for the user to switch between applications, since different applications are controlled in more or less the same way.

GEOS also changes what is possible to do with the c64. Having a built-in diskTurbo system makes possible applications which are much more data intensive. Data base and other applications may incorporate much larger amounts of data. The scope of programs possible on the c64 increases.

Learning any new system is an investment in time. From the very beginning though, the amount of time and energy put into learning GEOS should pay rewards in the ease of implementing features in your program that would otherwise take much longer. The goals of GEOS are simple: greater utility and performance for the user, greater utility and simplicity for the programmer. This manual is part of our effort in achieving these goals.

Speaking the Same Language

Before we begin, a word about the notations which we'll use is in order. Within this manual we refer to constants, memory locations, variables, and routines by their symbolic names. This makes for much easier reading than trying to remember a thousand different hexadecimal addresses. A jsr DoMenu is much more descriptive than a jsr \$78EC. The actual addresses and values for the symbolic names may be found in the appendices Memory_map, Constants, and Routines. As a convention, constants are all in upper

case (FALSE, TRUE), variables begin lower case and have every following word part capitalized (mouseXPosition, mouseData) and routine names have every word part capitalized (DoMenu). In addition to using symbolic names, we also use some simple assembler macros. For example,

```
LoadB      variable,value
```

is just a macro for

```
lda        #value
sta        variable
```

A complete listing of the macros used in GEOS appears in the Appendix.

The Basics

The following features are supported by GEOS and are described in this manual:

- Pull-down menus
- Icons
- Proportionally spaced fonts
- String I/O routines using proportionally spaced fonts
- Dialog boxes
- Complete graphics library
- Complete math library
- Multitasking within applications
- Fast disk access
- Paged file system
- Complete set of printer interfaces

GEOS is a full-fledged operating system, and its central part is the Kernal. The Kernal is a memory resident program, i.e., it is always in the c64 memory and is running all the time. It is the Kernal that contains support for all the windows, menus, icons, fonts and other features of GEOS. The deskTop, on the other hand, is not a part of the GEOS Kernal but is an application just like geoWrite and geoPaint. In fact, one could write an en-

tirely different file manipulation "shell", as such programs are called, and throw away the deskTop altogether.

Much of the programming under GEOS consists of constructing tables to define menus and icons and specifying routines for the Kernal to call when the menus and icons are activated. GEOS is a full operating system. It works like this.

Any input the user can send to an application running under the GEOS Kernal - pulling open a menu, activating a menu, entering text, moving the mouse - is called an event. The GEOS Kernal provides the support for processing events. The application supplies a table to define the menus, icons, and other events as well as a service routine to be executed when the event is activated by user input. When the GEOS Kernal determines an event has occurred it calls* the appropriate service routine. Service routines may then make use of GEOS text, graphics, disk turbo, or other routines to implement the action desired.

Applications may still have direct control over the hardware, but in many cases much of this support can be ceded to the Kernal.

As an example, instead of passing a signal to the application like "the mouse was clicked," the GEOS Kernal might conclude from several mouse movements and clicks that a menu event has occurred, i.e., a menu was pulled down and a selection was made. Routines inside the GEOS Kernal called dispatchers figure out what the user is doing, whether it be a menu, icon, or other event, and calls the proper user defined service routine to handle it.

In the case of our menu event above, the GEOS Kernal would reverse video flash the selected menu box and call the proper service routine provided for the activated menu selection. This type of interaction is known as event driven programming.

* In this manual unless otherwise noted, calling or dispatching refers to executing a jsr (jump to subroutine), and return refers to executing an rts (return to subroutine).

An event is defined as

1. a user initiated action or
2. a user defined time based process.

An example of a process would be a routine which is run every second to update a clock. The application programmer provides the routine and tells the GEOS Kernal how often to run it. Every time that amount of time elapses an event is triggered. When there are no user actions taking place only the GEOS Kernal code is running. Most applications can run entirely event driven. The GEOS Kernal supports moving the mouse, and detecting whether the mouse button is clicked over an icon, a menu, or some other area on the screen. The memory location `otherPressVector` contains the address of a routine to call when the user clicks the mouse outside any menu or icon. The memory location `keyVector` contains the address of a routine to call when a key on the keyboard is hit. The application may then call a routine that returns all buffered input. In an application such as an editor, the screen represents part of a page. Clicking the mouse in the screen area has the meaning of selecting a position on the page. This position then becomes the position at which to enter text or draw graphics.

When the user clicks the mouse in the screen area (outside of menus or icons), the routine whose address is stored in `otherPressVector` is called. The routine may look at the variables `mouseXPosition` and `mouseYPosition` to determine the position of the mouse. When a key, or keys are hit, the routine in `keyVector` is called and the application may then call `getNextChar` to return the characters entered by the user. `otherPressVector` and `keyVector` are initialized to 0 indicating there are no routines to call. The application's initialization code should set these vectors to the address of appropriate routines or leave them 0 if no service routine is being provided.

Double Clicks through `otherPressVector`

Double clicking is clicking the mouse button quickly twice in succession. The reader is already familiar with double clicking an application's file icon on the deskTop to cause the application to be run. Here we discuss double clicking through `otherPressVector`. Double clicking on an icon is discussed in the icon chapter.

The GEOS Kernal supports a variable called `dblClickCount`. To support a double click we do the following. The first time the mouse is clicked over the screen area, the `otherPressVector` routine is dispatched. As part of the service routine we check the value of `dblClickCount` and if it is 0, load it with the constant `CLICK_COUNT`. Our service routine then does anything else it needs to do to service a single click, and return. Every interrupt, `dblClickCount` is decremented if it is not already 0. If the screen area is clicked on again before `dblClickCount` has reached 0, then our service routine will know that this is the second of two clicks and may take the appropriate action.

Together with `otherPressVector` and `keyVector`, the menu and icon service routines provide the tools to design most simple applications. To provide even more flexibility, the GEOS Kernal makes provisions for running non-event routines for applications needing them. These will be described later.

Getting Started

The first thing an application should do when run from the GEOS deskTop is to define its menus, icons, and indicate the service routines to call for keyboard input and mouse presses. It should also clear the screen and draw any graphic shapes it needs to set up the general screen appearance.

When a user double clicks on an application's icon from the deskTop, the GEOS Kernal will initialize the system to a default state, load the application, and perform a jsr to the application's initialization routine. The address of the initialization routine is specified in the application's File Header block, which we'll describe later. The initialization routine contains data tables for defining the menus, icons, and other events, and calls GEOS routines for reading the tables and setting up the events. It also draws the initial screen. Upon completion, the initialization routine returns to the GEOS Kernal. The main program loop in the GEOS Kernal will now be running and will be ready to handle menu selections, icon presses or any other event defined by the application.

When any event is triggered, the GEOS Kernal calls the service routine specified by the application. Just as the initialization routine did, each service routine executes and returns to the GEOS Kernal.

Summary

Several important points have been covered in this section. To summarize, the GEOS Kernal is a operating system which shares the memory space of the c64 with an application and is running all the time. The GEOS Kernal handles much of the low-level hardware interaction. When an event occurs, such as the keyboard being pressed, or a menu being selected, the GEOS Kernal calls the proper application service routine as specified in the applications's initialization code. The application service routine processes the event, possibly calling upon GEOS graphics and text support routines, and eventually returns to the GEOS Kernal. The GEOS Kernal is then ready to process the next event and dispatch the proper service routine.

When the application's icon is double clicked by the user, the GEOS Kernal loads the application, initializes the system to a default state, and calls the application's initialization routine. The initialization routine provides the necessary tables and calls the proper GEOS Kernal routines for setting up the application's events. It also draws the initial screen.

In this manual we explain exactly how all this is done and show examples of menus, icons, and text input in a small sample application. Used in this capacity an application may be easily prototyped in a week. To give a more intuitive idea of how the GEOS Kernal works, we describe its overall structure in the next section.

The GEOS Kernal Structure

There are two levels of code running within the GEOS Kernal, MainLoop and InterruptLevel.

MainLoop

The GEOS Kernal MainLoop is just one long loop of code. It checks for events and dispatches the proper application service routine. Each time it goes through its cycle, the MainLoop code checks for any user input and determines its significance.

A mouse button click can signify:

- an icon being selected,
- a menu being opened,
- a item being selected from an open menu,
- or, outside of any menu or icon, an activation of otherPressVector.

Keyboard input generates:

user entered text to be dealt with by an application's keyVector service routine, or text for a dialog box to be processed by the GEOS Kernal.

A process timeout signifies:

that an application service routine should run.

Given the input, MainLoop decides what to do. In the case of a menu, for example, it will figure out if

1. a submenu needs to be pulled down, e.g., the edit menu is selected and edit menu choices need to be displayed, or
2. an item that triggers a service routine is being selected, e.g., Cut under the edit submenu is selected and the application service routine to cut text needs to be run.

InterruptLevel

The GEOS Kernal InterruptLevel code handles the 6510 IRQ interrupt which is triggered 60 times a second by a raster interrupt on the c64. Every 60th of a second, the processor is stopped in its execution of MainLoop, and the InterruptLevel code is run. InterruptLevel completes in much less than a 60th of a second. All it does is read the hardware. Thus even if MainLoop takes much longer than a 60th of a second (by executing very long application service routines, for example), InterruptLevel will maintain a timely interaction with the hardware: Keys pressed on the keyboard or clicks of the mouse button won't be lost.

InterruptLevel saves the state of the machine and goes about interacting directly with the hardware. It buffers keyboard input, decrements the process timers (see the section on processes), moves the sprites and mouse, and detects presses of the mouse button. For example, if the mouse button is pressed, InterruptLevel sets a flag that is checked by MainLoop. MainLoop decides what to do depending on whether the mouse was positioned over a menu, icon, or somewhere else on screen. Thus, the first part of an event sequence always starts in InterruptLevel. Processes, the mouse, and the keyboard are watched by InterruptLevel and when changes are detected flags are set which MainLoop checks at least once each time through its loop. InterruptLevel restores the state of the machine when it exits and returns to MainLoop. MainLoop processes any changes detected in InterruptLevel and calls the appropriate application service routines.

Most c64 programmers are used to writing their own MainLoop and InterruptLevel code. It is important to realize that this is already done by the GEOS Kernal. The GEOS Kernal is akin to a skeleton that the programmer fleshed out. GEOS compatible applications consist of a collection of tables for defining events and service routines to handle the events. The flow of control is structured by the Kernal.

Whenever a service routine returns, it returns to MainLoop. Any service routine may redraw the screen, entirely reinitialize all events, new icons, menus and anything else, and safely return to the MainLoop. MainLoop will then continue where it left off, just after the call to the service routine. Thus a menu item can be defined that causes the application to go to another "screen" with all new functions. The service routine for this menu item may erase the screen and initialize new menus and icons. When the menu item service routine returns to MainLoop, MainLoop will continue checking for events, but will be checking the newly defined ones. Usually the next event to check for is an icon press. If a menu was selected, however, MainLoop will skip the icon check since an icon and a menu could not have both been selected with the same press. The same is true with other event checks. During the next MainLoop, the new menus, icons, and other events will be checked.

Letting the GEOS Kernal do much of the dirty work and having the application define and process events, frees the applications programmer from having to reinvent the wheel everytime. This approach is sufficient to program even complex applications. geoWrite, geoPaint, and the deskTop were programmed in this fashion. To make programming even easier, the GEOS Kernal provides many utility routines (graphics, text, disk) that aid application development. The following section covers how to call the GEOS Kernal routines.

Calling GEOS Kernal Routines

This section gives a brief description of how the GEOS Kernal routines are used by the programmer. This should make the following programming examples clear. The first convention adopted when we began to develop the GEOS Kernal was to set aside some variable storage in zero page (zpage). This was done because 6502 instructions use less space and execute quicker in zpage. We also made the convention that the GEOS Kernal routines would use this variable space to accept parameters, perform internal calculations, and return values. Making routines modular like this with specific input and output makes it easier to track how each routine changes memory, and also makes it easier for developers other than Berkeley Softworks to use the GEOS Kernal routines.

To this end, 30 bytes in zpage beginning at location 2 are set aside for use as pseudoregisters. These memory locations divided into 15 word length variables with the names r0, r1, r2, ..., r15. The low byte of each pseudo register may be referenced as either rN or rNL, where N is the number of the register: e.g., r0, r0L. The high bytes may be individually referenced as rNH, e.g., r1H, r0H.

Typically, arguments to the GEOS Kernal routines are passed and returned in these pseudoregisters. This way all the GEOS Kernal routines may perform all their internal calculations with zpage variables. Instead of starting off trying to manage hundreds of the GEOS Kernal locations in your head, the programmer starts off with only fifteen.

The pseudoregisters are not the only way to pass parameters to the GEOS Kernal routines. Sometimes a, x, y, and even the carry flag are used for speed. There is also another way known as an in-line call. An in-line call solves the problem that when a routine is used frequently, a large number of bytes within an application can be taken up simply by the assembly language instructions that load the pseudo registers for the routines with the proper values. Some frequently used routines therefore have an in-line form to save bytes. Whereas normally a routine gets its parameters from pseudoregisters, the in-line version will get its parameters from the bytes immediately following the call to the routine. For example, the in-line call to the routine to draw a rectangle is shown below.

```
jsr      i_Rectangle ; draw a rectangle in the current system pattern. (The
                                ; system patterns can be changed with the routine
                                ; SetPattern.)
.byte    0            ; top of rectangle. Possible range: 0 - 199
```


.byte	199	; bottom of rectangle. Possible range: 0 - 199
.word	0	; left side. Possible range: 0 - 319
.word	319	; right side. Possible range: 0 - 319

Whereas the standard call looks like:

LoadB	r2L, 0	;top of rectangle. Possible range: 0 - 199
LoadB	r2H,199	;bottom of rectangle. Possible range: 0 - 199
LoadW	r3,0	;left side. Possible range: 0 - 319
LoadW	r4,319	;right side. Possible range: 0 - 319
jsr	Rectangle	;draw it

When an in-line routine is called, the first thing it does is to pop a word off the stack. Instead of pointing to the return address though, this word points to the parameters passed in-line after the jsr. The in-line routine picks up its parameters, loads the proper pseudoregisters with them, stuffs the correct return address back on the stack, and then enters the regular routine.

In-line routines make sense when a routine is called a large number of times with fixed values, such as Rectangle. A call to a `i_Rectangle` to erase or set up part of an application screen within an application works well with an in-line call since the input parameters don't change. It takes fewer bytes to store parameters as `.byte` and `.word` immediately following the subroutine call and have the subroutine include the code to pick the values up then it does to include the code to load the proper pseudoregisters before each call to the routine. To be more specific, a `LoadW r3,0` takes up 8 bytes whereas a `.word 0` only takes up only two. In-line routine names always begin with a `i_`.

Utility routines taking several fixed arguments have in-line entry points. Other routines less frequently called, or requiring only 1 or 2 parameters do not have an in-line form.

In this section we talked about how applications call GEOS utility routines, and how the GEOS Kernal calls user routines in response to events. We covered `MainLoop`, and `Interrupt Level` code within the GEOS Kernal and what each is responsible for. In the next section we cover how an application may include its own code directly within `InterruptLevel` or `MainLoop`. Generally this is not recommended, but in some circumstances, like supporting special external hardware, it may be required. When this is necessary, the application can load special vectors provided in system RAM that allow the addition of code to `InterruptLevel`

or MainLoop. Most programmers may skip the next paragraph on non-event code. A good rule of thumb is to avoid altering MainLoop or InterruptLevel code. In particular, an application specific interrupt routine can lead to difficult to fix synchronization bugs between MainLoop and InterruptLevel code.

Non-Event Code

Most applications will never need non-event driven code. This is code that needs to run every interrupt or every MainLoop regardless of what the user is doing and also cannot be set up as a process. The only cause for this is supporting a special hardware device. The programmer who needs to run non-event triggered code may do so by altering certain system vectors provided for that purpose. The vectors for adding interrupt and MainLoop code are interruptTopVector, interruptBottomVector, and applicationMain. If an application has interrupt code it wants executed before the GEOS Kernal Interrupt Level code, it can alter the address contained in interruptTopVector.* An indirect jump is performed through interruptTop Vector which normally contains the address of InterruptMain.

Putting the address of an application routine here will cause it to be run at the beginning of each interrupt. The end of the application's interrupt routine should contain a jmp to InterruptMain. Similarly, to execute code after normal the GEOS Kernal Interrupt Code has run, alter interruptBottomVector. At the end of InterruptMain code, the GEOS Kernal does a subroutine call to the address contained in interruptBottomVector unless it is \$0 (its default value). Any routine executed through interruptBottomVector should perform an rts, not rti upon completion.

Most programming can be accomplished through events. Additional MainLoop routines can be added, however, by loading applicationMain with the address of the routine to call. During each MainLoop a jmp indirect is made through applicationMain unless it is zero (its default value.) Performing an rts at the end of the routine called through applicationMain will return properly to the GEOS Kernal MainLoop.

* For the actual address values of RAM variables double check with Memory_map in the Appendices. The locations of RAM variables may change between this writing and product release and its quite easy to miss updating an address value buried in the text. An actual assembler listing of the Memory_map and all RAM variables as well as the addresses of user callable GEOS Kernal routines will be included.

Steps in Designing a GEOS Application

We can now breakdown what is involved in programming under GEOS.

Choose the events:

decide what menus, icons, etc. the application is to have. A special kind of event is a time base process which we will cover in a later chapter.

Define the events:

load the vectors or construct the tables which define the events themselves. For example, menu structures are defined with a simple table structure.

Write the routines:

construct the routines which are called by MainLoop to service the events you've defined.

To this point, this first section aims to provide an overview of what programming under the GEOS Kernal is like. GEOS allows an application to be very quickly prototyped because it break the program up into smaller easier to tackle event definition tables and event service routines. Before we begin coding the events for the application we present a short discussion of the hardware setup used by GEOS: the graphics mode it uses, its layout in memory, and how the bank-switching registers are set.

It is actually possible to program under GEOS and not know anything about graphics modes or bank switching, so if you are new to the c64, don't worry if this next section seems difficult. It assumes you have read the Commodore 64 Programmer's Reference Guide. It is unlikely that you will need to change the standard GEOS memory map. However, you may on occassion wish to access a favorite routine in the Comodore Kernal ROM, or a floating point routine in the BASIC ROM and then return to normal execution. The remainder of this chapter is devoted to a "physical" description of GEOS. That is, the grapics mode its programmed in, where it is located in memory, how to tell what version Kernal is running, what the hardware control register are set to and how to alter the memory map to use Kernal or BASIC ROM routines.

Hi-Resolution Bit-Mapped Mode

GEOS uses the bit-mapped graphics mode of the c64 at a resolution of 320 by 200 pixels. In this mode, 8000 bytes (200 scanlines by 40 bytes per line) are used to display the screen. If you are unfamiliar with this mode you may want to refer to the Commodore 64 Programmer's Reference Guide (see page 121 for a general description of the hi-resolution bit-mapped graphics mode as well as pages 102 to 105 for some useful tables) .

To make programming applications under the GEOS Kernal easier, another 8000 byte buffer is kept which is usually used to hold a backup copy of the screen data. Routines are provided which copy the image stored in the background buffer to the screen (foreground buffer) and vice versa. This is helpful when a menu is pulled down over the application's window, or a dialogue box appears, and it writes over the data on the foreground screen. To recover what was on the screen previously, the menus and dialog boxes copy the background screen to the foreground screen thus saving the application the trouble of having to recreate the screen itself, something which sometimes is impossible.

These recovery routines are accessible from application routines as well. The geoPaint application uses these routines to "undo" graphics changes which the user decides to discard. the GEOS Kernal routines used to recover from background include, RecoverAllMenus, RecoverLine, RecoverMenu, and RecoverRectangle. These routines are explained in the graphics and Menu sections of this manual. Buffering to the background can be disabled if the application's program desires to use the area in the background buffer for some other purpose such as for expanding available code space. This is also described in the graphics section under Display Buffering.

MemoryMap

The GEOS Kernal Memory Map table documents the c64 memory used by the GEOS Kernal and that which is left free for use by the application. Applications have about 22k from address \$0400 to \$5FFF. With special provision, applications may also expand over the background screen buffer. This opens up another 8k bringing the total to about 30k. This may seem like a limited amount of memory at first, but it is important to realize that all the menu, icon, dialog box, disk, file system, and various buffer support is included within the GEOS Kernal. This means much less work for the developer, less expensive development,

shorter product cycles and it also means that the 22k to 30k left to the developer will go a lot further. The speed of the disk access routines also make it practical to swap functional units in and out during program execution. Very large and sophisticated applications can be developed using memory overlay techniques. In fact the new GEOS VLIR file structure as described in a later chapter is designed to facilitate loading program modules into memory as needed.

The location of application code and RAM is all that most developers will ever need to know about the GEOS Kernal memory map. RAM is provided in three separate places, plus whatever application space the programmer wants to devote to it. First, the pseudo-registers r0 - r15 may be used by applications. GEOS routines also use these locations. The registers used by each GEOS routine are well documented. Second, there are 4 bytes from \$00FC to 00FF in zpage that are unused by either BASIC or the c64 Kernal. These are used as pseudoregisters a0 and a1. By passing values to utility routines in zpage locations and having them use these zpage pseudoregisters internally, a large number of bytes can be saved because zpage locations only generate one byte of addressing. This far outweighs the bytes wasted loading and unloading the pseudoregisters with parameters before and after each routine call.

Another zpage area is provided, from 70 - 7F. These are the pseudoregisters a2 - a9. Finally, the memory area from \$7F40 - 7FFF is available for non-zpage RAM. For a complete variable layout, see the memory map in the Appendix.

GEOS MEMORY MAP

Num. Bytes Decimal	Address Range Hexadecimal	Description
1	0000	6510 Data Direction Register
2	0001	6510 I/O register
110	0002-006F	zpage used by GEOS and application
16	0070-007F	zpage for only application, regs a2-a9
123	0080-00FA	zpage used by c64 Kernal & BASIC
4	00FC-00FE	zpage for only applications, regs a0-a1
1	00FF	Used by Kernal ROM & BASIC routines
256	0100-01FF	6510 stack
512	0200-03FF	RAM used by c64 Kernal ROM routines
23552	0400-5FFF	Application program and data
8000	6000-7F3F	Background screen RAM
192	7F40-7FFF	Application RAM
2560	8000-89FF	GEOS disk buffers and variable RAM
512	8A00-8BFF	Sprite picture data
1000	8C00-8FD7	Video color matrix
16	8FD8-8FF7	GEOS RAM
8	8FF8-8FFF	Sprite pointers
4096	9000-9FFF	GEOS code
8000	A000-BF3F	Foreground screen RAM or BASIC ROM
192	BF40-BFFF	GEOS tables
4288	C000-CFFF	4k GEOS Kernal code, always resident
4096	D000-DFFF	4k GEOS Kernal or 4k c64 I/O space
7808	E000-FE74	8k GEOS Kernal or 8k c64 Kernal ROM
378	FE80-FFF9	Input driver
6	FFFA-FFFF	6510 NMI, IRQ, and reset vectors

All I/O, screen drawing and interrupt control can and should be handled by the GEOS Kernal. The Kernal routines are extremely easy to use and take up memory space whether the application uses them or not. The following section describes in detail the hardware configuration used by the GEOS Kernal and can be skipped by most users. If, for example, you plan on supporting an I/O device which the GEOS Kernal does not (yet) support, or will be writing in BASIC instead of assembler, this material will be relevant.

GEOS Kernal Version Bytes

There are several bytes within the GEOS Kernal that identify what version GEOS is running. At location \$C006 we find the string "GEOS BOOT". This string can be used to determine if the application was booted from GEOS. Developers who will not be using the GEOS Kernal routines in their applications can write over all but \$C000 to C07F which are used to return the user to the deskTop after quitting the application. These bytes may be copied elsewhere and moved back to reboot GEOS. Adding an icon to an application is easiest done with the Icon Editor available on GEOS deskPack 1.

Immediately following the "GEOS BOOT" string are two digits containing the version number. Currently these bytes may be \$12 or \$13 for versions 1.2 or 1.3, respectively. For GEOS Kernals version 1.3 and beyond have additional information bytes just after the version byte. First there is a language byte. Following the language byte are three bytes that are reserved for future expansion and are currently \$0. As of this writing, the English, German, French, Dutch, and Italian have been implemented, whereas the Swedish, Spanish and Portuguese language versions have not.

This area appears in memory as shown on the following page.

GEOS Kernal Information Bytes

```

.psect      $C000                ;Kernal code starts at $C000

BootGEOS:
    jmp      o_BootGEOS          ; Jump vector back into GEOS.  If the
                                ; routine o_BootGEOS moves in future versions
                                ; of GEOS, doing a jmp to BootGEOS at $C000 will
                                ; still work.  As long as the space
                                ; $C000 to $C02F is preserved, a jump to
                                ; $C000 will reboot GEOS

ResetHandle:
    .jmp      internal routine    ; This is a jump vector used by the
                                ; internals of GEOS

BootFilename:
    .byte     "GEOS BOOT"        ; This is at $C006. This string can be used
                                ; to check if an application was booted
                                ; from GEOS.

VERSION:
    .byte     $13                ; A hex byte containing the GEOS version
                                ; number.  The current version is 1.2

;FOREIGN language byte:

                                ;ENGLISH          =    0
                                ;GERMAN            =    1
                                ;FRENCH            =    2
                                ;DUTCH             =    3
                                ;ITALIAN           =    4
                                ;SWEDISH           =    5      (not implemented)
                                ;SPANISH           =    6      (not implemented)
                                ;PORTUGUESE        =    7      (not implemented)

.if ENGLISH
    .byte     0                  ; ENGLISH
.endif
.if GERMAN
    .byte     1                  ; GERMAN
.endif
.if FRENCH
    .byte     2                  ; FRENCH
.endif
.if DUTCH
    .byte     3                  ; DUTCH
.endif
    .byte     0                  ;Reserved for future use
    .byte     0                  ;Reserved for future use
    .byte     0                  ;Reserved for future use

```


Bank Switching and Configuring

The major part of the GEOS Kernal occupies memory from \$BF40 on up. This means that the GEOS Kernal is using RAM in address space which is normally used for other purposes. The address space from D000 to DFFF is normally used as I/O space, but the c64 has RAM which can be swapped in over this area. Similarly, the c64 Kernal ROM and BASIC ROM can be bank switched out and another 8k of RAM opened up. During normal operation, all the GEOS Kernal banks are swapped in and the BASIC, c64 Kernal ROM, and I/O space are mapped out. All I/O processing is handled by the GEOS Kernal during interrupt level and the GEOS Kernal takes care of all the bank switching itself.

The selected bank is determined by the contents of location \$0001 and two lines coming from the cartridge and external ROM ports. Since the GEOS Kernal runs without any ROM cartridges, the internal pull up resistors on these two cartridge lines cause them to default to high. The placement of screen RAM and the ROM character set is determined by the contents of address \$D018.

If your application needs to access I/O space outside of the GEOS Kernal routines, or access the c64 Kernal or BASIC ROMS, it should make use of two GEOS Kernal routines, `InitForIO` and `DoneWithIO`. These routines will take care of changing and restoring the memory map, and disabling interrupts and sprites as needed.

Memory mapping is described in the Commodore 64 Programmer's Reference Guide (pages 101 through 106 and 260 through 267). On the following is a table which outlines the default settings which the GEOS Kernal uses.

GEOS Control Register Settings			
Control Function	Memory Location	Value Stored	Description
Bank Select	0001	xxxx000x	Selects which ROM banks to appear in the address space. GEOS swaps c64 kernal, I/O and BASIC out.
VIC Chip Location Select	DD00	xxxx010x	Chooses which 16k address range the Vic chip can address. GEOS selects bank 2 at \$8000 - \$BFFF
Screen Memory	Top 4 bits of D018	1000xxxx	Together with the VIC chip bank select, determines the location of the video RAM. GEOS uses A000 - BF3F
Char Memory	Bits 1, 2, 3 of D018	xxxx010x	Chooses where c64 character ROM will appear in the selected bank. GEOS puts it at \$9000-\$97FF, so that it doesn't interfere with the video RAM at A000-BF3F

Constants for RAM/ROM Bank Switching			
IO_IN	=	\$35	; 60K RAM, 4K I/O space in
RAM_64K	=	\$30	; 64K RAM system
KRNL_BAS_IO_IN	=	\$37	; both Kernal and BASIC ROMs in
KRNL_IO_IN	=	\$36	; ROM Kernal and I/O space mapped in

Assembler Directives

Our development environment here at Berkeley Softworks may not be similar to yours. The assembler we use is of our own design. In the sample application presented throughout this manual then the reader will be seeing our assemblers directives and our macros. We will then try to keep the usage of macros to a minimum and will try to provide list file outputs when necessary. Below is a table listing the assembler directives or pseudo-operations as they are sometimes known .

Assembler Directives Used in Examples

Type	Pseudo Op Directive	Arguments	Description
Set Address:			
	<code>.psect</code>	<code>[VALUE]</code>	VALUE is used as an address. Following code is assembled starting at address VALUE. If value is missing, starts a relocatable section which the linker locate.
	<code>.ramsect</code>	<code>[VALUE]</code>	Same as psect except for variable RAM.
Label:			
	<code>NAME:</code>		Assigns the current address to NAME. Colon is optional if the label is not indented.
Constants:			
	<code>NAME</code>	<code>=[key]VALUE</code>	Equate NAME to VALUE, where VALUE is a decimal number unless preceded by a key character: '\$', indicates hex and % is binary.
Data:	<code>.byte</code>	<code>val1, val2, ...</code>	Store val1, val2, ... in sequential bytes.
	<code>.word</code>	<code>val1, val2, ...</code>	Store val1, val2, ... in sequential 16 bit words.
RAM:	<code>.block</code>	<code>VALUE</code>	Allocates value number of sequential bytes. Usually follows a label.
Conditional:			
	<code>.if</code>	<code>expression</code>	<code>if</code> expression is true assemble the enclosed program code. <code>elif</code> ends an if section and begins another. <code>else</code> ends an if and begins an alternate section. <code>endif</code> terminates any open structure.
	<code>.elif</code>		
	<code>.else</code>		
	<code>.endif</code>		

What's to Come

In the following sections we show exactly what programming under GEOS is like by building the first part of a sample application using a simple initialization routine which clears the screen and puts up a few icons. Each of the icons when activated will dispatch a service routine which will again clear the screen and then put up a different menu structure. We describe how icons and menus are defined under the GEOS Kernal.

After this we include a short section that explains one way of getting your test application onto disk and running it with GEOS.

In later sections we will add graphics, text, file handling, dialog boxes, processes, printing, and sprite support in the same manner. Each section consists of a general explanation, followed by an example, and finishes with a complete description of the callable GEOS Kernal routines.

After all this, we present tutorials on how to write input and printer drivers and cover the various library routines. When testing features such as icons and menus, it is often useful to use dummy service routines that merely execute an rts. This way menu and icon structures can be tested and verified before adding true service routines. After these events are defined, menus will pull down and icon structures will blink even though they will merely call empty service routines. This allows the structure of the program to be tested and verified before the actual code is written.

2.

Icons and Menus

Icons

The reader is assumed to have used GEOS and is familiar with menus and icons. We will not, therefore, include an in depth discussion on what they are and how they act. In short, an icon causes a routine to be called when the mouse is clicked over its graphic. Menus are similar except that one may go through several levels before opening a menu with the item whose routine is to be run. Icons and menus are mostly large table structures. The easiest way to learn how to create them is by example. We, therefore, begin our discussion by constructing a table for icons.

Icons are usually initialized by the applications initialization routine. As mentioned in the first chapter, upon the double click of an application's icon, GEOS will perform a jsr to the application's initialization routine as specified in the application's File Header. The sample application presented here begins at \$400. All it does is call `i_Rectangle` (described in the

section on graphics) to clear the screen and then calls the icon initialization routine, DoIcons.

The programmer designs an Icon Table that contains all the necessary information for the GEOS Kernal to display and operate the icons. He passes the address of the table to the routine DoIcons.

```
LoadW      r0,QuantumTest Icons      ;address of icon data structure
jsr        DoIcons                    ;put up a few icons
```

The Icon Table first indicates the number of icons to define and where to position the mouse after the icons have been drawn. Our sample application sets up eight icons. The icon table is comprised of one seven byte entry for each icon to be defined.

Icon Figure 1 - Icon Table

```
Y_POS_TOP_ICON = 10      ; arrange icons relative to upper left icon
X_POS_TOP_ICON = 3        ; x pos is in bytes

QuantumTestIcons:
    .byte    8              ; number of icons
    .word    160            ; x pos to place mouse after icons displayed
    .byte    100           ; y pos to place mouse after icons displayed

                                ;ENTRY 1
    .word    showCaseData   ; ptr to graphic data for showCase
    .byte    X_POS_TOP_ICON ; x byte pos. of top left corner of icon
    .byte    Y_POS_TOP_ICON ; y pixel pos. of top left corner
    .byte    2, 16         ; width in bytes and height in pixels
    .word    DoShow        ; service routine for showCase

                                ;ENTRY 2
    .word    justForFunData ; ptr to graphic data for hand
    .byte    X_POS_TOP_ICON ; x byte pos. of top left corner
    .byte    Y_POS_TOP_ICON+30 ; y pixel pos. of top left corner
    .byte    2, 16         ; width in bytes and height in pixels
    .word    DoFun         ; service routine for hand

    ...                    ; 6 more icon entries
```

Each icon entry begins with a pointer to the graphic data for the icon. The graphic data for each of the icons in the sample application is 33 bytes long. The next two entries

contain the x and y position to place the upper left corner of the icon on screen. Finally, there is a pointer to the service routine to call when the icon is pressed. In the sample application, all the icons point to the same service routine.

Below we have the bit-mapped data for the icon pictures and the service routines (routine in our case) themselves.

Icon Figure 2 - Service Routines

```

DoFun:                ; service routine for Just For Fun icon
DoShow:               ; service routine for Commodore Software
                      ; Showcase icon
...                   ; remaining icon service labels

jsr    i_Rectangle
        .byte    0
        .byte    199
        .word    0
        .word    319

LoadW   r0,Screen2Icon    ; address of icon data structure
jsr     DoIcons           ; put up icon for quitting to the deskTop
                      ; must have at least one icon, GEOS bug.
lda     #0                ; place mouse cursor on first menu selection
LoadW   r0,QuantumTestMen ; address of menu structure
jsr     DoMenu            ; put up a menu

rts

```

All of the service routine labels are located at this one routine. Icon service routines can do just about anything. They can even reinitialize the entire system. Icon service routines can also react to both single and double clicks. Supporting double clicks for icons is easier than for otherPressVector routines. The GEOS Kernel passes TRUE (\$FF) to indicate a double click and FALSE (0) to indicate a single click in r0H. r0L is passed with the number of the icon as it appears in the icon table. The first icon in the table is icon 0. This is useful if several icons are using the same service routine, but need to have a slightly different effect depending on which icon was selected. The service routine can check which icon number is activated by its number in r0L and act accordingly.

Our service routine in Icon Figure 2 above calls `i_Rectangle` to erase the screen and then calls `DoIcons` to initialize a new table of icons. This new icon table defines just one icon that quits to the `deskTop` with a call to `EnterDeskTop`. In addition, this service routine also sets up a call to `DoMenu` to initialize a menu structure for the "second screen" of the sample application. We leave the discussion of menus till the next section.

Lastly, we define the graphics data for the icons pointed to from the icon definition table.

Icon Figure 3 - Data Table

```
showCaseData:
    .byte    $02,$FF,$9C,$80,$01,$80,$39,$80,$6D,$80,$E5,$81,$BD,$83,$19,$86
    .byte    $31,$8C,$61,$98,$C1,$B1,$81,$BB,$01,$BE,$01,$BC,$01,$80,$01,$02
    .byte    $FF

justForFunData:
    .byte    $02,$FF,$9C,$87,$01,$8D,$81,$9E,$C1,$BB,$61,$AD,$B1,$B6,$99,$9B
    .byte    $09,$9D,$0D,$96,$07,$9A,$03,$8C,$01,$87,$E1,$80,$31,$80,$19,$02
    .byte    $FF
    ...
                                ; remaining graphics data for 6 more
                                ; icons
```

Icon pictures are bit-mapped graphics. These icons are 2 bytes wide and 16 scanlines high. The data is in `BitmapUp` compacted format. (See `BitmapUp` in the graphics section for a description of this format.) The bit-maps for these icons do not compact very well because they use one more bytes than a straight bit mapped does. (Compaction can be very efficient. For example the entire ruler in `geoWrite` takes up 16 bytes.) The entire icon definition table is shown in unabbreviated form in the next section.

There is a technique used by the `deskTop` and `geoPaint` that is worth explaining here. In the `deskTop`, when user clicks once on a file icon, it inverts to show it is selected. The file icon stays inverted until another icon is selected. The user may then pull open a menu and select an item: for example, `Get Info`. The `Get Info` routine then gets information for the file whose icon is inverted. A similar interaction happens in `geoPaint` when a drawing tool is selected. It stays inverted until a new tool is selected. Selecting the second icon causes the

first to uninvert, and the second to invert. How is this communication between an icon and a menu in the first case, and between two icons in the second accomplished?

There is a variable in the GEOS Kernel called `iconSelFlag`. Bits 6 and 7 control how icon selection is indicated to the user. If bit 7 is set, the icon is flashed in reverse video and back indicating the icon was selected. There is a constant for this, `ST_FLASH = $80`, so you don't have to try and remember which bit it is. The delay between flashes is set by the variable `selectionFlash`. The default value for this byte is `SELECTION_DELAY = 10`.

If bit 7 is clear and bit 6 is set, then the icon will be inverted reverse video. The constant for this value is `ST_INVERT = $40`. In this case it is up to the icon service routine to save the location and number, and any other information about the file icon by storing values in application variables. When a menu item is selected, it may check the values of the variables left by the icon service routine and take the appropriate action. In the Get Info example above, the service routine will reinvert the icon with a call to `InvertRectangle`, and retrieves the Get Info information from the file's File Header Block.

If both bits 6 and 7 in `alphaFlag` are clear nothing will be done to the icon graphic to indicate it has been selected.

One important note before we leave icons: GEOS assumes that an application will always be using at least one icon. If you find your application has no use for icons, it is a good idea to create one anyway. Make it one byte wide and one scanline high, and make its pointer to graphics data equal to 0. This way you will not have to include a dummy picture and the icon will never get called.

There is only one routine for icons, `DoIcons` as described below.

Dolcons

Function: Draw and turn on icons as defined in an Icon Table.

Pass: r0 - pointer to Icon Table

Inline: .word - pointer to Icon Table

Return: nothing

Destroyed: a, x, y, r0 - r11

Synopsis: Dolcons displays the icons as defined in the Icon Table pointed to by r0, (or by the .word trailing the jsr Dolcons). When displayed, the user may select any icon by pressing the mouse while over the icon picture and the service routine indicated in the Icon Table will be executed.

To turn an icon off, change its picture pointer to \$00,00. If the icon was already active then you will have to erase it from the screen. If its picture pointer is zeroed before the icons are drawn, it will never appear.

The only way to reliably turn a whole set of icons off is to erase the screen, and initialize a new set of icons, or to turn all icons off by setting their picture pointers to 0.

The GEOS Kernal passes TRUE (\$FF) to indicate a double click on the icon and FALSE (0) to indicate a single click in r0H. The Kernal also passes r0L with the number of the icon as it appears in the icon table. The first icon in the table is icon 0. This is useful if several icons are using the same service routine, but need to have a slightly different effect depending on which icon was selected. The service routine can check which icon number is activated by its number in r0L and act accordingly.

Menus

Menus, like icons, are mostly table structures, and the best way to learn how to create them is by example. The first thing to do is set up a call like the following:

```
jsr i_DoMenu          ;put up menu  
.word TestMenu        ;pointer to menu definition table structure
```

Just like icons, menu structures are specified in a table. The sample application shows the complete menu structure we will cover piece by piece.

Menus can be vertical or horizontal. To put up the menu bar at the top of the screen, we use a horizontal menu. The first four entries of the definition table are the top, bottom, left and right sides of the vertical menu stretched out to the right. The top of the menu is usually set near the top of the screen and the bottom is computed from the top by adding 14, the height of menu items in the standard character set.

The right side of a horizontal menu is found by experimentation. When creating a menu for the first time, choose a value far enough right to accomodate all the menu items, and then add some extra to be safe. Rather than trying to guess where the right side of the menu will lie, use 319 as the right side of the menu, test the menu, and then adjust it to the left. Strange bugs happen if the menu box does not have enough room for the menu keywords.

GEOS draws one long box for the menu, the dimensions passed in the table. It then prints the menu names inside it, one after the other on the same line, separating each one with the vertical bar character, '|'. The vertical bar divides the long box into what then appears to be separate boxes for each item. If the value chosen for the right side is too far to the right, the last item will simply have excess blank space between its last letter and the right side of the long menu box. This is easy to readjust.

The next entry is the menu type, horizontal or vertical, OR'ed with the number of menu items. Thus these first five entries look like this:

Menu Figure 1

```

MAIN_TOP  =      10      ;constants for the box dimensions
MAIN_BOT  =      24      ;near top of screen
MAIN_LFT  =       0      ;one line high for horizontal menu
MAIN_RT   =     255      ;very left side of screen
                        ;found by experimentation.

```

QuantumTestMenu:

```

.byte MAIN_TOP      ;top of menu
.byte MAIN_BOT      ;bottom of menu
.word MAIN_LFT      ;left side
.word MAIN_RT       ;right side
.byte HORIZONTAL     ;menu type OR'ed with number of menu items

```

Menu Selections

The next entries describe the data associated with each selection appearing in the menu. Each menu selection requires three item pieces of information. The first is a pointer to the text string containing the word(s) to display in the menu box. The second item is a byte indicating the type of the menu selection. Menu selections may open up a submenu or cause a service routine to be dispatched. Each of the menu selections along the top menu bar cause a submenu to be expanded. (These submenus however, will call service routines.) The third item is a pointer to either the submenu structure itself or a pointer to the service routine. The two horizontal menu items in our sample are:

Menu Figure 2

```

.byte ChangeDeptText ; pointer to null terminated text string
.byte SUB_MENU       ; flag indicating that this item opens up to
                        ; a submenu
.byte ChangDeptMenu  ; the pointer to the menu structure
.byte ShowcaseText   ; pointer to null terminated text string
.byte SUB_MENU       ; flag indicating that this item opens up to a
                        ; submenu
.byte ShowcaseMenu    ; the pointer to the menu structure

```

What follows now is the two text strings pointed to from the table above:

```

ChangeDeptText:      .byte          "Change Departments",0
                        ; text for this item
ShowcaseText:        .byte          "Commodore Software Showcase ",0
                        ; text for this item

```

The two submenu structures for the Change Departments and Commodore Software Showcase are next. The complete example appears in the next section. Since menus are repetitive structures, we will print one or two examples of each type here and save the complete structure for the sample application listing in the next section.

A submenu structure is similar to the main menu structure. First we decide what values to use for the position of the submenu so that it appears at the right place in relation to the main menu structure when it opens up. This is largely a matter of experimentation. The first two entries in the Change Departments submenu show that this is a VERTICAL submenu which causes a MENU_ACTION instead of opening out to another level of submenu.* The third through eighth menu entries are the same three line definitions as the Showcase and Fun items and are unshown here. They define menu actions for the remaining menu selections under the ChangeDeptMenu main menu.

Menu Figure 3

```

ChangeDeptMenu:
    .byte  MAIN_BOT           ; top of menu starts at bottom of main menu
    .byte  MAIN_BOT+8*14+1    ; bottom of menu allow 15 for each line
    .word  MAIN_LFT           ; left side
    .word  MAIN_LFT+155       ; same right side as main
    .byte  VERTICAL|8         ; menu type | number of menu items
    .word  ShowcaseText       ; text for this item
    .byte  MENU_ACTION        ; flag to indicate a submenu
    .word  ShowcaseDsp        ; address of submenu structure
    .word  FunText            ; text for this item
    .byte  MENU_ACTION
    .word  FunDsp
    ...                      ; 6 more three line items like the two above

```

*It is possible to link vertical and horizontal submenus to an arbitrary depth, having a horizontal menu selection open a vertical menu which can then open up another horizontal menu and so on.

This submenu contains 8 menu selections of which two are shown. The remaining 8 are similar. The top of a vertical menu is usually positioned at the bottom of the horizontal menu which spawned it. Each menu selection will be 14 pixels high. GEOS will add a final single pixel line at the bottom. The bottom of the menu can be computed as,

$$\text{MenuBottom} = \text{MenuTop} + (\text{NumOfItems} * 14) + 1$$

This is about the only menu dimension you can easily derive without experimentation. This submenu is given the same left side as the main menu and a right side found purely by experimentation. The first menu selection will display the text pointed to by ShowcaseText, and when activated will cause a MENU_ACTION: the routine ShowcaseDsp, pointed to by the next memory word, will be executed.

The remaining menu selection items are similar. Following the ChangeDepartMenu menu is the ShowcaseMenu. This menu is the same as the ChangeDepartMenu except that it uses a constrained menu. The source for this menu can be found within the complete listing for the sample applicatin in the next section. The joystick pointer will not go outside the boundries of a constrained menu after it has been opened. This option is useful for allowing the user to quickly move to the bottom of the menu without having to worry about falling off the end and having to start all over. This works well for very long menus or three level menus in which a horizontal menu opens up from a vertical menu. In the latter case, it is easy to slip off the top or bottom of the menu when moving horizontally to select an item. The constrained indicator is a bit in the menu type byte:

```
.byte    CONSTRAINED|VERTICAL|7 ; menu type, and number of items
        ; Constrained menus keep joystick on menu.
```

Following the ShowcaseMenu are the 15 text strings for selection of the two sub menus. These text strings are not reprinted here but can be found in the listing of the sample in the next section. What remains are the dispatch routines for the menu actions. These dispatches may do anything but should include a call to one of the menu routines like GotoFirstMenu which will roll up and deactivate the open menu structures. The dispatches for the sample application don't do anything except call GotoFirstMenu.

```

LearningDsp:                                ;Change Departments Dispatches
FunDsp:
ServiceDsp:
InfoNetDsp:
PeopleConDsp:
NewsDsp:
MallDsp:

CatalogDsp:                                ;Software Showcase Dispatches
PreviewDsp:
SIGLibDsp:
FileTransDsp:
ReviewDsp:
PostOfficeDsp:
ChangeDsp:
ShowcaseDsp:

    jsr GotoFirstMenu                        ;roll menus back up

    rts

```

That's all there is to a simple menu structure. There are several possible variations, though. There is a routine that can be used in place of GotoFirstMenu, called DoPreviousMenu which will only roll up one level of menus. This can be used to allow the user to set two or more options under the same menu. Thus, suppose you have a vertical menu with selections A, B, C, and D. Each of these four selections may expand a horizontal menu with items 1, 2, 3, and 4. Your user wants to be able to simply set option 1 under D and 3 under C. When the user selects the 1 under D instead of rolling all the windows up, call RecoverMenu to roll up only the horizontal menu. Your service routine will also have to load mouseXPosition and mouseYPosition to a point on the vertical menu, or as soon as the Kernal rolls up the horizontal menu it will "pull it out from under" the mouse and the Kernal it will think instead that the user has moved the mouse off the menu and will roll up the remaining vertical menu.

Menu routines can be as clever as desired. They can even be self-modifying, though this is usually not a good idea to try at first. For example an asterisk can be placed next to currently active options such as bold or italic from a style menu in a word processor. Everytime a new style is chosen, the service routine goes into the menu table itself and writes the asterisk into (or erases it from if deselecting the style) the text string for the menu item.

There is another menu type called DYNAMIC_SUB_MENU which allows you to specify a routine to be called before the menu is actually unfolded. The word which follows the .byte DYNAMIC_SUB_MENU is a pointer to a routine to call before the submenu is opened.

Normally this word points to the submenu structure, so before it exits, the dynamic submenu routine should load r0 with the address of the submenu structure to open.

Dynamic submenus are useful for building a menu structure on the fly. The routine may check the state of the system and alter the menu table before the menu is displayed. In the text chapter of this manual an approach is discussed for creating the point size menu (as appears in geoWrite and geoPaint) when a font is selected. When the font is selected, the dynamic submenu routine checks to see which point sizes are available and builds out the point size submenu table. The various routines and options for menu support including DYNAMIC-SUB_MENU are summarized in the following.

Menu Constants

Menu Type Values

HORIZONTAL

Indicates that the items of the menu will be arranged horizontally:

Item 1	Item 2	Item 3
--------	--------	--------

VERTICAL

Indicates that the items of the menu will be arranged vertically:

Item 1
Item 2
Item 3

CONSTRAINED

Indicates that mouse is constrained to the area of the menu. Mouse can still be moved off the top of the menu to cancel the menu and avoid any menu action.

Menu Selection Types

SUB_MENU

Indicates that this menu selection opens a submenu structure. It is followed by a pointer to the submenu structure.

MENU_ACTION

Indicates that this menu item causes a dispatch routine to be called. It is followed by a pointer to the routine.

DYNAMIC_SUB_MENU

Indicates that this menu item causes a dispatch routine to be called before a submenu is displayed. It is followed by a pointer to the routine. The routine should leave in r0 the address of the submenu structure.

DoMenu

Function: Display and activate the menu structure pointed to by r0 (or the inline .word.)

Pass: a - number of menu selection to place cursor on
r0 - address to menu table

Inline: .word menuTable

Return: nothing

Destroyed: r0 - r13, a, x, y destroyed

Synopsis: Given the menu definition table, DoMenu draws the menu and takes care of all menu processing. As long as the menu table is correct, DoMenu will support nested menus to an arbitrary depth. After the menu is initialized by the call to DoMenu, menu processing is handled by the GEOS Kernal in MainLoop. Clicking the mouse over a menu item, and opening a submenu or calling a service routine is all handled by the Kernal.

DoMenu is usually called from an application's initialization code. (GEOS performs a jump to \$400 after loading the application. The initialization code for the application is usually found here including a call to DoMenu.)

ReDoMenu

Function: Re-enable a menu after an item has been selected so that another selection may be made.

Pass: nothing

Return: nothing

Destroyed: r0 - r13, a, x, y destroyed

Synopsis: The menu handling code in MainLoop knows about the structure of the menus being displayed on screen. When a menu selection of type `MENU_ACTION` is activated, the menu program calls an application supplied service routine. As part of this routine, the programmer must tell GEOS what to do with the currently extended menu. `ReDoMenu` re-enables the extended menu so that another selection can be made from it.

DoPreviousMenu

Function: Remove a submenu and re-enable the previous menu and the mouse

Pass: nothing

Return: nothing

Destroyed: r0 - r13, a, x, y destroyed

Synopsis: The menu handling code in the GEOS Kernal MainLoop knows about the structure of the menus being displayed on screen. When a menu selection of type `MENU_ACTION` is activated, the menu program calls an application supplied service routine. As part of this routine, the programmer must tell GEOS what to do with the currently extended menu.

DoPreviousMenu will erase the currently extended submenu and re-enable the previous one. The mouse is enabled so that a selection from the previous menu may be activated. When using DoPreviousMenu, make sure to reposition the mouse over the previous menu before the call to DoPreviousMenu or else after the menu is draw the MainLoop Kernal code running the menus will,

1. detect the mouse outside the menu;
2. assume the user has move the mouse off a menu in order to cancel it; and
3. erase all extended menus.

GotoFirstMenu

Function: Roll up all submenus. Called from menu service routine.

Pass: nothing

Return: nothing

Destroyed: r0 - r13, a, x, y destroyed

Synopsis: When a menu selection of type MENU_ACTION is activated, the menu handling code in the Kernal MainLoop calls an application supplied service routine. As part of this routine, the programmer must tell GEOS what to do with the currently extended menu.

GotoFirstMenu rolls up all the currently open submenus and reverts back to the main menu.

Getting Up and Running

We now know enough to create a simple application that uses icons and menus. The next step is getting the sample application up and running. First, we need to make GEOS symbol information — routine addresses, constants, variables — available to the application.

Second, is a more sticky problem: GEOS files are different from standard c64 files. Currently existing assembler/editors don't know about GEOS file structures, and cannot save them to disk. We must figure out some way of saving the application as a GEOS file.

To solve this problem, we use an approach that allows us to create our program as a normal Commodore PRG file. We then run a simple BASIC program on it that turns it into a GEOS APPLICATION file. Lets do this now, before presenting the file system chapter, in order to get a small application up and running as quickly as possible. Since with any new operating system there is a bit of a chicken and egg problem in having to know everything before being able to do anything. For now then, be content with the fact that we do not as yet fully understand the GEOS file structures.

The idea of this chapter is to make sure all problems with running GEOS in your development environment are solved before beginning serious programming.

Including Constants, Memory_Map, Routines

The sample application at the end of this section was produced with our assembler. The advantage of publishing it in this form is that you can see exactly what code actually ran on the c64. The sample application file starts with a series of lines that begin with `.include filename`. An `.include` directive indicates that text from another file is to be inserted. Here the files being included are `Constants`, `Memory_map`, and `Routines`. `Constants` contains the constant definitions as you might expect, `Memory_map` defines the variable labels and buffers, and `Routines` contains the jump table, the addresses to call for each of the GEOS routines.

These three files are included in the Appendix. The first thing you will have to do in order to run your programs is to type these files into your system. Once you've done that it is a good idea to make several copies. If your assembler will only take one source file, then you

will have to copy all three files into one file along with your program code. You may find that you have to shorten the variable names if your assembler only recognizes 8 or 9 characters.

Once that laborious task is done, you may start patterning your sample application after our sample application. A good idea might be to start with our actual sample application, or some subset of it, and make the formatting changes necessary to run with your assembler.

Structuring the Application

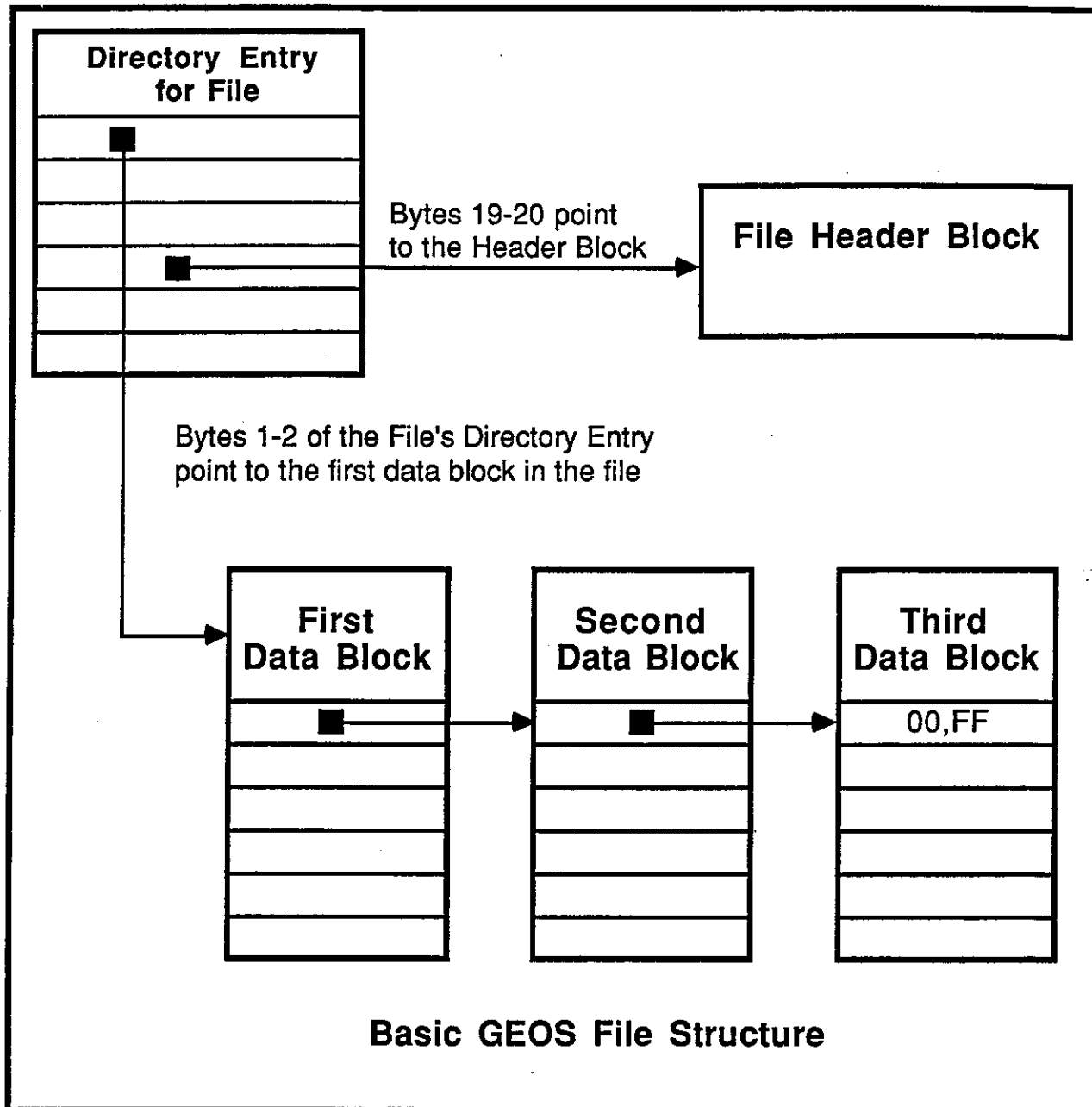
GEOS files contain an extra block called the File Header block. It contains the bit maps for the icon, and other information that GEOS needs to know about the file such as the load address and execution address. The load address is the address where the application should be loaded into memory. The execution address is the address of the initialization routine to be called by the GEOS Kernal when the application is double clicked on the deskTop.

Unfortunately, there are currently no assembler/editors that can save a GEOS file to disk. They don't know about the File Header block. There are two ways to overcome this. The first is to use GEOS routines to save the application to disk. Unfortunately, since GEOS initializes memory before it runs, you would need to boot GEOS and then somehow break it in mid execution. If you could then figure out a way of getting your application into memory you could include with your code calls to GEOS routines to save it to the disk. One of the ways in which this can be done is by using special hardware called In-Circuit-Emulator (ICE) unit.

Fortunately though, there is another way. This involves making your assembler think that it is saving a normal c64 PRG file. The File Header block is included as the first block in the file. A BASIC program then unlinks the File Header block, puts it in its proper place and reconnects the remainder of the program. Before we go any further, let's look at the what the proper place for a File Header block is.

File Header Block

The diagram below shows the structure of a standard GEOS file. As is usual with a c64 file, the file's Directory Entry points to the first data block in the file. What is different is that it also points to the File Header block. In a regular c64 file these bytes would point to the first side sector in a REL file. Since GEOS doesn't use relative files, these bytes are free to point to the Header Block.



The trick to getting your assembler/editor to save a GEOS file to disk is to include the File Header block as the first block in the file. Since the memory space available to GEOS programs begins at \$400, we edit the application to include the Header block beginning at \$400 as data. Since the first block from a file on disk contains a track and sector pointer to the next block in the file in its first two bytes and a load address in its second two bytes, there is only 252 actual data bytes in the file. Therefore, the application should be located at $\$400 + 252 = \$4FC$. When saved to disk, the application code will begin right at the beginning of the second block.

The load, and execution addresses in the File Header block are both set to \$4FC. (This assumes that the initialization routine is the first routine in the program. If it isn't, then the execution address should be changed accordingly.) We then assemble and save the application to disk as a normal c64 PRG file. It appears on the deskTop as a normal c64 icon. The Header block is saved to the first sector on disk, and the application code in the remainder.

The BASIC program presented below will transform the file into a GEOS file. Bytes 19 and 20 in the Directory Entry will point to the Header block, and bytes 1 and 2 will point to the first block in the application. After transformation, the file's custom icon appears on the deskTop. Double clicking on the icon will cause the GEOS Kernal to load the application at the load address as specified in the File Header block and begin execution at the execution address, also specified in the File Header block. This means that the application will be loaded at \$4FC. The first 252 bytes available in the file will be "wasted." There are two ways around this.

The first is the easiest: use the first 252 bytes (and more if you want) as RAM. The second is to,

1. indicate \$400 for the load and execute addresses in the Header block,
2. save the file to disk,
3. convert the file to the GEOS format, and
4. re-edit with your assembler editor.

The bytes in the Directory Entry that are different for GEOS files will probably not affect your assembler/editor; you may still be able to use it to edit the converted file. However, your assembler/editor may change bytes in the Directory Entry that GEOS needs, specifically the pointer to the File Header block. If the only bytes your assembler/editor changes are for the filename, and number of blocks in the file, then you're ok. After loading the file with your assembler/editor you can simply move the file to \$400, or start editing it there if all you saved originally was a dummy block. If the above doesn't work, you will have to edit an unconverted copy of your file (with the Header block as the first block), save, and convert your file every time you want to test it.

In the next section we present a skeleton structure for the unconverted file and the BASIC program to convert it into a GEOS file. The File Header is the first block, followed by a couple of dummy statements where the program would be. We'll explain the File Header as we go along. The complete File Header description can be found in the file system chapter.



3.

GEOS File in PRG Format

Introduction

The sample code below begins at \$400 with the GEOS File Header as data. The Header block is only 252 bytes long. The first four bytes of the File Header block as it will appear on disk are missing below, because four bytes will be inserted by the assembler/editor at the beginning of the block when the file is saved to disk. The first two of these bytes are added because the first two bytes in every block on disk (for both c64 and GEOS files) contain either a pointer to the track and sector of the next block in the file, or \$00 as a track number which signifies the last block in the file, followed by a byte containing an index from the beginning of the block to the last byte in the file. When the Header block is saved to disk, these two bytes will point to the next block in the file. When our BASIC program converts the file to a GEOS file it will change these two bytes to \$00,\$FF, indicating no following blocks and all bytes in the block used.

The third and fourth bytes are added to the block on disk because the assembler/editor thinks this is a PRG file and in a PRG file these bytes contain the load address for the file. GEOS doesn't want these bytes tacked on the front of the File Header because it stores the start address elsewhere. Luckily, these bytes should instead contain the dimensions of the programs icon which are always 3 and 21. It is a simple matter for the conversion program to overwrite these bytes with 3 and 21.

The next byte in the test sample is the number of bytes in the icon picture with the top bit set. (The top bit is set as a flag to the GEOS routine that prints the icon on the deskTop.) Sixty three bytes of icon data follow. After the icon is the Commodore file type which is USR (All GEOS file appear as USR files to the c64) and the GEOS file type which should be APPLICATION. After that is the GEOS file structure type, which should be SEQUENTIAL, and the load and start addresses for the application. The filename is next, followed by the author' name. The complete File Header structure and what its elements signify is described in the file system section of this manual.

.psect \$400

FileHeader:

;The first four bytes as they will appear on disk after file is
;converted. These bytes are not be present in memory and so are
;commented out.

```
;.word    $00,$FF      ;pointer to filename string
;.byte    3             ;icon is 3 bytes wide
;.byte    21            ;and 21 lines high
```

```
.byte    (63+$80)      ;64 bytes of icon picture info
.byte    %11111111,%11111111,%11111111
.byte    %10000000,%00000000,%00000001
.byte    %10000000,%00000000,%00000001
.byte    %10000000,%00000000,%00000001
.byte    %10011111,%11110000,%00000001
.byte    %10000001,%00000000,%00000001
.byte    %10000001,%00000000,%00000001
.byte    %10000001,%00011110,%00000001
.byte    %10000001,%00100001,%00000001
.byte    %10000001,%00101110,%00000001
.byte    %10000001,%00100000,%00000001
.byte    %10000001,%00011110,%00000001
.byte    %10000001,%00000000,%00000001
.byte    %10000001,%00111100,%00000001
.byte    %10000001,%00100010,%00000001
.byte    %10000001,%00011000,%01111101
.byte    %10000001,%01000100,%00010001
.byte    %10000000,%00111100,%00010001
.byte    %10000000,%00000000,%00010001
.byte    %10000000,%00000000,%00000001
.byte    %11111111,%11111111,%11111111
.byte    $80|USER      ;Commodore file type assigned to GEOS files
.byte    APPLICATION   ;GEOS file type
.byte    SEQUENTIAL    ;normal, sequential file structure
.word    $400          ;start address for saving file data
.word    EndCode       ;end address for saving file data
.word    $400          ;address to jump to after loading application
.byte    "Test Appl  V1.0",0,0,0,0      ;16 byte filename + 4 0's
.byte    "Mike Farr",0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ;16 byte author name + 4 0's
.byte    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ;16 bytes for parent appl. name + 4 0's
.byte    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ;ignore for the present
.byte    0,0,0,0      ;
```

InitCode:

;The code here will be stored in the second block on disk. The
code for the application should go here. The address of InitCode
is \$4FC.

```
lda #0                ;dummy statements here for second block
sta #0
...
```

TestApplication

Called By: Called from deskTop

Synopsis: This file contains a skeleton test application. It is designed to be entered with any standard assembler/editor, or even as BASIC data statements. The first 252 bytes of this file contain the File Header for the application and will appear as the first block in the file when saved on disk. The file will be converted after it is stored into a GEOS file by a separate BASIC program.

 This program will disconnect the File Header from the chain of blocks in the file and write its track and sector to the proper pointer in the Directory Entry. The track and sector pointer in the Directory Entry for the first data block will be changed to point to the first code block which was stored after the File Header.

PRGTOGEOS

The BASIC program for converting our test application to a GEOS application appears below. It prompts for the name of the file, and the date. GEOS uses bytes 23 - 27 in the Directory Entry to contain the last modification date for the file. Several other bytes from the Header File are transferred to the Directory Entry. The format of the Directory Header as modified by GEOS appears immediately below.

Directory Entry for Test File	
0	c64 file type: 0=DELETED,1=SEQUENTIAL,2=PROGRAM,3=USER,4=RELATIVE Bit 6=used as write-protect bit
1 2	Track and sector of first data block in this file
3 ... 18	16 character file name padded with shift spaces, \$A0
19-20	Track and sector of GEOS File Header Block
21	GEOS file structure type: 0=SEQUENTIAL
22	GEOS file types: 0=NOT_GEOS, 1=BASIC, 2=ASSEMBLY, 3=DATA, 4=SYSTEM, 5=DESK_ACC, 6=APPLICATION, 7=APPL_DATA, 8=FONT,9=PRINTER, 10=INPUT_DEVICE, 11=DISK_DEVICE
23	Date: year last modified, offset from 1900
24	Date: Month last modified (1 - 12)
25	Date: day last modified (1 - 31)
26	Date: hour last modified (0 - 23)
27	Date: minute last modified (0 - 59)
28-29	Number of blocks (sectors) in the file

PRGTOGEOS

Purpose: BASIC program for changing the test application from a PRG to GEOS file.

Synopsis: The first block of the test application as stored on disk contains the File Header for the application. This program converts the file into a GEOS file.

The Filename of the test application is prompted for and may be no longer that 15 characters, even though a GEOS filename may be up to 16 characters. Next the date to store for the file is prompted for and then the disk is searched for the file.

When the file is found, this program disconnects the File Header as the first block in the file and writes its track and sector to the proper pointer in the Directory Entry. The track and sector pointer in the Directory Entry for the first data block will be changed to point to the first code block which was stored as the second block in the file.

The first two bytes in the File Header Block are changed to \$00,FF and the second two are changed to 3,21 which is the width in bytes and height in scanlines of the file icon picture.

Bytes 21 and 22 in the Directory Entry are changed to be the GEOS file and structure types respectively, and the Date for the file inserted in bytes 23 - 27. The number of blocks in the file is decremented.

```

50      INPUT "PROGRAM FILENAME";F$           ; prompt for input filename
60      INPUT "YEAR" (EX: 86)";Y             ; prompt for time/date
70      INPUT "MONTH" (EX: 5)";MO
75      INPUT "DAY" (EX: 31)";DA
80      INPUT (EX: 14)";H
85      INPUT "MINUTE" (EX: 35)";MI
100     OPEN 15,8,15,"I:0"                     ; open a file
110     OPEN 2,8,2,"#"                         ;
120     TS=CHR$(18):SS=CHR$(1)                 ; track 18, sector 1= first dir block
122     GOSUB 1000                             ; read the dir block into the drive
124     GET#2,NT$,NS$                          ; get track and sector to nextdir block
125     FOR E=0 TO 7                          ; check each dir Entry
130     GOSUB 3000                             ; get a filename

```


[illegible]

```
3020      GET#2,B$:I=I+1                      ; READ FILENAME SUBRTINE
3030      IF ASC(B$)=160 GOTO 3050             ; if end of filename skip rest of entry
3040      D$=D$+B$:GOTO 3020                  ; get next file name char
3050      FOR I=I TO 31                        ; skip this dir entry
3060      GET#2,B$                             ; read a char
3070      NEXT I
3080      RETURN
```

Getting It to Run

With a little elbow grease, and a lot typing, the programmer should be able to get a small application, like our sample application, to assemble and run on the Commodore. It is important at this stage of the game that the programmer stop and practice getting a file to the disk and convert them to GEOS file. A good approach is to get a very small sample application running and then exercise all the features which GEOS has to offer little bit at a time.

The following sections each cover a functional unit. After each section, the programmer may add to the sample application, so that by the time we have finished our tour of GEOS all aspects of GEOS will be familiar. We resume next with graphics.

This test is designed as demonstration of menus and icons.

[illegible]

```
.if (0)
```

Quantum Link GEOS Demo File, Main Routine

Author: Mike Farr

Called By: Set up as an application, Load from deskTop

Pass: Nothing

Return: Nothing

Destroyed: Assume all registers

Synopsis: This module contains routines designed to test menus and icons. When the routine is double clicked, execution begins at the label QuantumTest. It calls the appropriate routines to display the first screen.

The first screen shows 8 icons (corresponding to the eight items of the Q-link main menu). The icon pictures may be any size, these small icons were stolen from geoPaint and are admittedly under-dramatic for an initial title screen.

All the icons on the first screen execute the same dispatch routine. When any icon is activated, the dispatch routine erases the screen and draws the Main Menu structure of the second screen.

The second screen shows a menu structure as it might appear in Q-link. Names for the menu selections could be anything. All the menu selections point to the same dispatch routine. It just does the standard GotoFirstMenu that's needed to roll up the menus.

The icon in the lower right corner clears the screen and returns the user to the first screen.

```
.endif
```

```
.if(0)
```

QuantumTest

```
.endif
```

```
QuantumTest:                ;main routine activated from deskTop

    jsr    NewDisk           ;a bug with older versions of GEOS made it
                             ;necessary to call NewDisk to stop the disk
                             ;motor if the application does not access the
                             ;disk
    jsr    MouseUp           ;activate mouse, redundant if mouse
                             ;already active
    lda    #2
    jsr    SetPattern        ;set the system pattern to the 50% stipple
    jsr    i_Rectangle       ;clear the screen to system background pattern
    .byte  0
    .byte  199
    .word  0
    .word  319

    LoadW  r0,QuantumTestIcons ;address of icon data structure
    jsr    DoIcons           ;put up a few icons
    rts
```

```
.if(0)
```

First Screen Icon Structure

Called By: Initial QuantumTest routine through Dolcons

Pass: nothing

Return: nothing

Synopsis: The icon tables for the 8 icons on the first screen of the demo.

```
.endif
```

```
Y_POS_TOP_ICON = 10           ;position of icon is measured from
                                upper left corner.
```

```
X_POS_TOP_ICON = 3
```

QuantumTestIcons:

```
.byte    8           ; number of icons
.word    160          ; x pos to place mouse after icons displayed
.byte    100          ; y pos to place mouse after icons displayed

.word    showCaseData ; ptr to graphic data for showCase
.byte    X_POS_TOP_ICON ; x byte pos. of top left corner of icon
.byte    Y_POS_TOP_ICON ; y pixel pos. of top left corner
.byte    2, 16         ; width in bytes and height in pixels
.word    DoShow        ; dispatch routine for showCase

.word    justForFunData ; ptr to graphic data for hand
.byte    X_POS_TOP_ICON ; x byte pos. of top left corner
.byte    Y_POS_TOP_ICON+30 ; y pixel pos. of top left corner
.byte    2, 16         ; width in bytes and height in pixels
.word    DoFun          ; dispatch routine for hand

.word    custServData  ; ptr to graphic data for eraser
.byte    X_POS_TOP_ICON ; x byte pos. of top left corner
.byte    Y_POS_TOP_ICON+60 ; y pixel pos. of top left corner
.byte    2, 16         ; width in bytes and height in pixels
.word    DoServ        ; dispatch routine for eraser

.word    mallData      ; ptr to graphic data
.byte    18             ; x byte pos. of top left corner
.byte    Y_POS_TOP_ICON+60 ; y pixel pos. of top left corner
```

```

.byte    2, 16                ; width  in bytes and height in pixels
.word    DoMall                ; dispatch routine for icon

.word    newsData              ; ptr to graphic data
.byte    33                    ; x byte pos. of top left corner
.byte    Y_POS_TOP_ICON+60     ; y pixel pos. of top left corner
.byte    2, 16                ; width  in bytes and height in pixels
.word    DoNews                ; dispatch routine for icon

.word    learningData          ; ptr to graphic data
.byte    33                    ; x byte pos. of top left corner
.byte    Y_POS_TOP_ICON+30     ; y pixel pos. of top left corner
.byte    2, 16                ; width  in bytes and height in pixels
.word    DoLearn               ; dispatch routine for icon

.word    cinData               ; ptr to graphic data
.byte    33                    ; x byte pos. of top left corner
.byte    Y_POS_TOP_ICON        ; y pixel pos. of top left corner
.byte    2, 16                ; width  in bytes and height in pixels
.word    DoCin                 ; dispatch routine for icon

.word    pConData              ; ptr to graphic data
.byte    18                    ; x byte pos. of top left corner
.byte    Y_POS_TOP_ICON        ; y pixel pos. of top left corner
.byte    2, 16                ; width  in bytes and height in pixels
.word    DoPCon                ; dispatch routine for icon

```

```

.if(0)

```

Dispatch Routines for Icons on Screen 1

Called By: Activation of icons on first screen

Pass: nothing

Return: nothing

Synopsis: Activating one of the icons on the first screen executes one of these routines. The all just clear the screen and load the menus of the second screen.

```

.endif

```

DoCin: ; dispatch routine for Commodore Information Network icon
DoLearn: ; dispatch routine for Learning Center icon

```

DoPCon:      ; dispatch routine for People Connection icon
DoNews:      ; dispatch routine for News and Information icon
DoMall:      ; dispatch routine for The Mall icon
DoFun:       ; dispatch routine for Just For Fun icon
DoServ:      ; dispatch routine for Customer Service Center icon
DoShow:      ; dispatch routine for Commodore Software Showcase icon

    jsr      i_Rectangle
    .byte    0
    .byte    199
    .word    0
    .word    319

    LoadW   r0,Screen2Icon      ;address of icon data structure
    jsr      DoIcons            ;put up a few icons
                                ;must have at least one icon, GEOS bug.
    lda      #0                 ;place mouse cursor on first menu selection
    LoadW   r0,QuantumTestMenu ;address of menu structure
    jsr      DoMenu             ;put up a menu

    rts

.if(0)

```

Graphics Data Tables for Icons

Synopsis: These icon graphics were stolen from the paint program.

```
.endif
```

```
.nodlist
```

```
showCaseData:
```

```

    .byte    $02,$FF,$9C,$80,$01,$80,$39,$80,$6D,$80,$E5,$81,$BD,$83,$19,$86
    .byte    $31,$8C,$61,$98,$C1,$B1,$81,$BB,$01,$BE,$01,$BC,$01,$80,$01,$02
    .byte    $FF

```

```
justForFunData:
```

```

    .byte    $02,$FF,$9C,$87,$01,$8D,$81,$9E,$C1,$BB,$61,$AD,$B1,$B6,$99,$9B
    .byte    $09,$9D,$0D,$96,$07,$9A,$03,$8C,$01,$87,$E1,$80,$31,$80,$19,$02
    .byte    $FF

```

```
custServData:
```

```

    .byte    $02,$FF,$9C,$80,$01,$80,$01,$BF,$01,$BF,$81,$BF,$C1,$AF,$E1,$A7
    .byte    $F1,$B3,$F1,$9A,$11,$8E,$11,$86,$11,$83,$F1,$80,$01,$80,$01,$02
    .byte    $FF

```

```
mallData:
```

```

    .byte    $02,$FF,$9C,$80,$01,$80,$01,$83,$81,$83,$81,$86,$C1,$86,$C1,$84
    .byte    $C1,$8C,$61,$88,$61,$8F,$E1,$98,$31,$90,$31,$B8,$79,$80,$01,$02
    .byte    $FF

```



```

newsData:
    .byte    $02,$FF,$8E,$80,$01,$80,$01,$90,$01,$98,$01,$8C,$01,$86,$01,$83
    .byte    $01,$02,$81,$8C,$80,$C1,$80,$61,$80,$31,$80,$19,$80,$09,$80,$01
    .byte    $02,$FF
learningData:
    .byte    $02,$FF,$9C,$80,$01,$80,$01,$9D,$B9,$90,$09,$90,$09,$80,$01,$90
    .byte    $09,$90,$09,$80,$01,$90,$09,$90,$09,$9D,$B9,$80,$01,$80,$01,$02
    .byte    $FF
cinData:
    .byte    $02,$FF,$9C,$80,$01,$80,$01,$9D,$B9,$90,$09,$90,$09,$80,$01,$90
    .byte    $09,$90,$09,$80,$01,$90,$09,$90,$09,$9D,$B9,$80,$01,$80,$01,$02
    .byte    $FF
pConData:
    .byte    $02,$FF,$9C,$80,$01,$80,$01,$9D,$B9,$90,$09,$90,$09,$80,$01,$90
    .byte    $09,$90,$09,$80,$01,$90,$09,$90,$09,$9D,$B9,$80,$01,$80,$01,$02
    .byte    $FF

.if (0)

```

Screen2Icon

Called By: The Dolcon call in the dispatch for the icons on the first screen.

Synopsis: Puts up the icon in the lower right corner of the screen for returning to the first screen.

```

.endif

```

```

Screen2Icon:
    .byte    1                ; number of icons
    .word    240              ; x pos to place mouse when activated?
    .byte    155              ; mouse y start position

    .word    showCaseData     ; reuse graphic data for showCase
    .byte    30               ; x byte pos. of top left corner
    .byte    150              ; y pixel pos. of top left corner
    .byte    2, 16            ; width in bytes and height in pixels
    .word    ReturnFirstScreen ; dispatch routine to return to first screen

ReturnFirstScreen:           ; dispatch for this icon
    jmp EnterDeskTop         ; return to the deskTop

```

```
.if(0)
```

Quantum Test Menu Structure

Called By: Quantum Demo Menu

Synopsis: The menu structure for the second screen. Contains "Change Departments" and "Commodore Software Showcase" menu selections.

```
.endif
```

```
MAIN_TOP    = 10
MAIN_BOT    = 24          ;one text line high horizontal menu
MAIN_LFT    = 0
MAIN_RT     = 255        ;found by experiment, still needs a little
tweaking
```

QuantumTestMenu:

```
.byte MAIN_TOP      ;top of menu
.byte MAIN_BOT      ;bottom of menu
.word  MAIN_LFT      ;left side
.word  MAIN_RT       ;right side
.byte  HORIZONTAL|2  ;menu type | number of menu items

.word  ChangeDeptText ;text for this item
.byte  SUB_MENU       ;flag to indicate a submenu
.word  ChangeDeptMenu

.word  ShowcaseText   ;text for this item
.byte  SUB_MENU       ;flag to indicate a submenu
.word  ShowcaseMenu
```

```
.if(0)
```

Text for Main Menu Selections

```
.endif
```

```
ChangeDeptText: .byte "Change Departments",0 ;text for this item
ShowcaseText:   .byte "Commodore Software Showcase ",0 ;text for this item
```

```
.if(0)
```

Change Departments Submenu

Called By: Submenu for "Change Departments" menu on second screen

Menu Items:	Commodore Software Showcase	The Mall
	Just For Fun	Commodore Information Network
	Customer Service Center	Learning Center
	People Connection	News and Information

```
.endif
```

ChangeDeptMenu:

```
.byte    MAIN_BOT                ;top of menu starts at bottom of main menu
.byte    MAIN_BOT+8*14+1         ;bottom of menu, allow 14 for each line+1 at end
.word    MAIN_LFT                ;left side
.word    MAIN_LFT+155            ;same right side as main
.byte    VERTICAL|8              ;menu type | number of menu items

.word    ShowcaseText            ;text for this item
.byte    MENU_ACTION             ;flag to indicate a submenu
.word    ShowcaseDsp             ;address of submenu structure

.word    FunText                 ;text for this item
.byte    MENU_ACTION             ;flag to indicate a submenu
.word    FunDsp                  ;address of submenu structure

.word    ServiceText             ;text for this item
.byte    MENU_ACTION             ;flag to indicate a submenu
.word    ServiceDsp             ;address of submenu structure

.word    PeopleConText           ;text for this item
.byte    MENU_ACTION             ;flag to indicate a submenu
.word    PeopleConDsp           ;address of submenu structure

.word    MallText                ;text for this item
.byte    MENU_ACTION             ;flag to indicate a submenu
.word    MallDsp                ;address of submenu structure

.word    InfoNetText            ;text for this item
.byte    MENU_ACTION             ;flag to indicate a submenu
.word    InfoNetDsp             ;address of submenu structure
```

```
.word LearningText      ;text for this item
.byte MENU_ACTION       ;flag to indicate a submenu
.word LearningDsp        ;address of submenu structure
```

```
.word NewsText          ;text for this item
.byte MENU_ACTION       ;flag to indicate a submenu
.word NewsDsp           ;address of submenu structure
```

```
.if(0)
```

Software Showcase Submenu Structure

Called By: Unfolded from main menu on second screen

Menu Items:

Software Catalog	Software Previews
SIG Software Library(+)	Person-to-Person File Transfer (+)
Software Reviews	Q-Link Post Office
Change To Another Department	

```
.endif
```

```
ShowcaseMenu:          ;submenu structure for CSS
.byte MAIN_BOT          ;top of submenu
.byte MAIN_BOT+(7*14)+1 ;bottom of submenu= top + (14 scanlines/text
                        ;line) * num textlines + one for bottom line
.word 94                ;left side of this vertical menu structure
.word 94+140            ;right side
.byte CONSTRAINED|VERTICAL|7 ;menu type and number of items. Constrained
                        ;means keep joystick pointer on menu

.word CatalogText       ;address of items text
.byte MENU_ACTION       ;indicates next word is dispatch routine
.word CatalogDsp        ;service routine for this menu item

.word PreviewText       ;address of items text
.byte MENU_ACTION       ;indicates next word is dispatch
.word PreviewDsp        ;service routine for this menu item

.word SIGLibText        ;address of items text
.byte MENU_ACTION       ;indicates next word is dispatch
.word SIGLibDsp         ;service routine for this menu item
```

```

.word   FileTransText    ;address of items text
.byte   MENU_ACTION      ;indicates next word is dispatch
.word   FileTransDsp     ;service routine for this menu item

.word   ReviewText       ;address of items text
.byte   MENU_ACTION      ;indicates next word is dispatch
.word   ReviewDsp        ;service routine for this menu item

.word   PostOfficeText   ;address of items text
.byte   MENU_ACTION      ;indicates next word is dispatch
.word   PostOfficeDsp    ;service routine for this menu item

.word   ChangeText       ;address of items text
.byte   MENU_ACTION      ;indicates next word is dispatch
.word   ChangeDsp        ;service routine for this menu item

```

```
.if(0)
```

Text for "Change Departments" Submenu Selections

```

LearningText: .byte   "Learning Center",0           ;text for this item
FunText:      .byte   "Just For Fun",0              ;text for this item
ServiceText:  .byte   "Customer Service Center",0   ;text for this item
InfoNetText:  .byte   "Commodore Information Network",0 ;text for this item
PeopleConText: .byte   "People Connection",0         ;text for this item
NewsText:     .byte   "News and Information",0       ;text for this item
MallText:     .byte   "The Mall",0                  ;text for this item

```

Text for "Commodore Showcase" Submenu Selections

```

CatalogText:  .byte   "Software Catalog",0
PreviewText:  .byte   "Software Previews",0
SIGLibText:   .byte   "SIG Software Library",0
FileTransText: .byte   "Person-to-Person file Transfer (+)",0
ReviewText:   .byte   "Software Reviews",0
PostOfficeText: .byte   "Q-Link Post Office",0
ChangeText:   .byte   "Change to Another Dept.",0

```

```
.endif
```

Dispatches for the "Software Showcase" and "Change Department" Submenu Selections

Called By: Change Departments Menu Commodore Software Showcase Menu

```
EndCode:                ;end of area saved in disk file
```

```
.if(0)
```

The Header Block for the Application

Synopsis: The Commodore 64 Programmer's Reference Guide discusses the Header Block, a block associated with each GEOS file containing information about the file. Among other things it contains start, and load addresses, the filename and the icon for the file.

If using the method described in the previous section for saving a file to disk as a regular c64 PRG file and converting it to a GEOS file with the PRGTOGEOS program, this Header Block should be the first block in the file. The first four bytes of this file are commented out as explained in that section.

```
.endif
```

```
GPadHdr:
```

```

; .word   $00FF           ;pointer to filename string
; .byte   3               ;icon is 3 bytes wide
; .byte   21              ;and 21 lines high
; .byte   (63+$80)        ;64 bytes of "Test" icon picture info
; .byte   %11111111,%11111111,%11111111
; .byte   %10000000,%00000000,%00000001
; .byte   %10000000,%00000000,%00000001
; .byte   %10000000,%00000000,%00000001
; .byte   %10011111,%11110000,%00000001
; .byte   %10000001,%00000000,%00000001
; .byte   %10000001,%00000000,%00000001
; .byte   %10000001,%00011110,%00000001
; .byte   %10000001,%00100001,%00000001
; .byte   %10000001,%00101110,%00000001
; .byte   %10000001,%00100000,%00000001
; .byte   %10000001,%00011110,%00000001
; .byte   %10000001,%00000000,%00000001
; .byte   %10000001,%00111100,%00000001
; .byte   %10000001,%00100010,%00000001
; .byte   %10000001,%00011000,%01111101
; .byte   %10000001,%01000100,%00010001
; .byte   %10000000,%00111100,%00010001
; .byte   %10000000,%00000000,%00010001
; .byte   %11111111,%11111111,%11111111
```

```
.byte    $80|USER           ;Commodore file type assigned to GEOS files
.byte    APPLICATION        ;GEOS file type
.byte    SEQUENTIAL         ;normal, sequential file structure

.word    $400               ;start address for saving file data
.word    EndCode            ;end address for saving file data
.word    $400               ;address to jump to after loading application
.byte    "quantum    V1.0",0,0,0,0      ;20 byte permanent name
.byte    "Mike Farr",0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ;20 byte author
.byte    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ;20 bytes for parent appl. name (not used)
.byte    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ;20 bytes for parent appl. name (not used)
.byte    0,0,0,0           ;20 bytes for parent appl. name (not used)
.end
```


4.

Drawing with GEOS

Drawing with Patterns

At this point the programmer should be able to put up a simple sample application and save it to disk. Now we can begin to add more features to it. In this chapter we explain how to use GEOS graphics. As mentioned in the first chapter, GEOS uses the high-resolution bit-mapped mode of the c64. The reader should read up on this mode in the Commodore 64 Programmer's Reference Guide if he is unfamiliar with it.

Drawing to the screen sets or resets bits in the Screen RAM. Set bits are displayed as the foreground color and reset bits the background color. The default foreground/background color combination under GEOS is dark grey for the foreground and light grey for the background. This may seem backwards since a blank screen is often thought of as black, but in GEOS the background represents a blank white sheet of paper and drawing is done in a black pencil. These colors may be changed by the preference manager so it is more useful to speak of background and foreground rather than drawing in one color or another.

Dimensions

All dimensions passed to GEOS graphics routine are inclusive: a line contains the coordinates passed as its endpoints, and a rectangle as described later, includes the lines that make up its sides, top and bottom.

Color

Due to the nature of color and memory constraints in the c64 several tradeoffs had to be made in graphics support of color. Currently, all the graphics drawing commands are done without affecting the hi-res bit-mapped mode color map. Thus, whenever lines or patterns are drawn, they will appear in the background and foreground colors. It is up to the application to manipulate the color map, and if necessary provide a 1000 byte background color map to allow undoing color changes. This is what the application geoPaint does.

Display Buffering

As mentioned in Chapter 1, GEOS keeps a second background 8000 byte buffer which is used to hold a copy of the contents of the regular foreground screen contents. The background buffer is used to recover the screen contents after a menu or other obstruction has been displayed on screen. It can also be used by applications to provide undo functions, screen buffering, or just as extra code space. To recover something from the background screen means to copy an indicated area from the background screen buffer to the foreground screen.

Menus, for example are drawn on the foreground screen only. After a menu item has been selected, the menu is erased by recovering the area underneath the menu to the foreground screen.

GEOS provides an option flag whereby all drawing, both graphic and textual, may be done to both the foreground and background buffers, to the foreground buffer only, or to the background buffer only. This flag is `displayBufferOn`. GEOS looks at bits 6 and 7 to determine how to draw to the screen:

`displayBufferOn`

bit 7 - write to foreground screen if set
bit 6 - write to background screen if set

The graphics routines like the line drawing routines discussed immediately below take into account the value of `displayBufferOn`. The most common use of `displayBufferOn` is to limit drawing of an object to the foreground screen so that it may be easily erased later by copying the background screen over it. Internally, this is what the GEOS Kernal does when it opens a menu or dialog box. The menu is drawn only on the foreground screen and when the menu needs to be erased, the background screen is copied over it. GEOS provides routines imprinting an object from the foreground screen to the background screen and for copying an object from the background screen to the foreground.

Some applications are so starved for memory that they decide to allocate the 8k background buffer to program code. The question arises here as to how to recover after Menus, Desk Accessories, and Dialog Boxes: there is no background buffer to recover from. In this case it is necessary for the application to provide its own routine for restoring the screen. There is a GEOS vector called `RecoverVector` which normally contains the address of the graphics routine `RecoverRectangle` (as described below). Whenever GEO recovers a Menu or Dialog Box it sets up parameters as if it were going to call `RecoverRectangle` and does a `jsr` to the routine whose address is in `RecoverVector`. If the application does not use screen buffering then it must load `RecoverVector` with the address of a routine that will redraw the screen.

If the user loads this word with the address of his own routine, then it will get called and have access to the same parameter information. This is what an applicatin does that uses the background screen for code. It must provide its own routine for redrawing the screen.

Drawing Lines

We now proceed to discussing the simplest form of drawing in GEOS, line drawing. Drawing vertical and horizontal lines are handled slightly differently from handling diagonal lines. Horizontal and vertical lines make use of a pattern byte. This enables the drawing of variously dotted lines. Diagonal lines are always drawn solid.

When drawing a horizontal line using a pattern byte, the pattern byte is always written directly to a byte in the screen RAM area. No shifting of data bytes is done. In order to have

the line start or stop on a position other than a byte boundary, bits are masked off either end of the pattern byte before it is written to the screen. The effect of this is that two lines drawn next to each other in the same pattern would match up. That is, even if the lines were of different length, all the zero and one bits in each pattern byte would be next to each other.

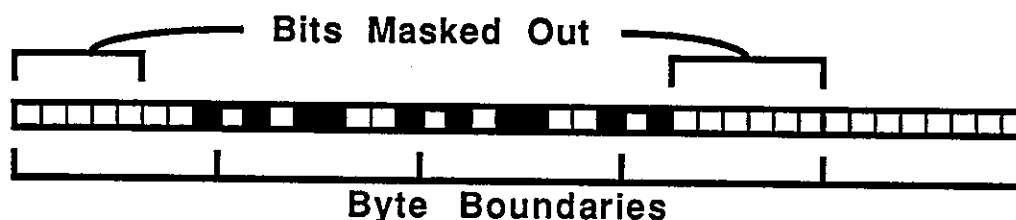
For example, suppose GEOS were to draw a line from point a to point b in the scanline:




with the given pattern byte:



The pattern byte would always be aligned so that bit zero in the pattern byte always appeared in bit zero of each byte in screen RAM. To make the line segment start and stop on the correct bits, the pattern would be masked to the proper number of bits when written to the end bytes of the line segment:



Vertical lines are drawn a similar way. In this case the pattern byte is turned vertically and written down the screen. Drawing vertical lines is a bit slow since when the pattern byte is "turned vertically" it is now being written over 8 different scanlines. This means that the zeroth bit is being written to the first byte, the first bit being written into the byte that appears immediately below the it, and so on. To write a single bit into a byte means OR'ing the bit into the byte. Drawing vertical lines is therefore at least eight times slower than drawing horizontal lines.



Diagonal Lines

Drawing diagonal lines is different from drawing horizontal or vertical lines. Diagonal lines are not drawn using a pattern byte. Instead they always draw a full line in which every bit is set. A diagonal line already appears somewhat jagged and drawing it in a pattern just makes it seem more jagged. There would also be the question of how to use the pattern byte, horizontally or vertically depending upon the slope of the line. The routines for drawing lines plus two routines dealing with points follow.

DrawPoint

Function: Draws a point in the foreground or background color, or recovers it from background screen buffer.

Accessed: If drawing, not recovering
displayBufferOn
- bit 7 - write to foreground screen if set
- bit 6 - write to background screen if set

Pass: sign flag - set for recover point from background, clear for drawing
carry flag - if sign flag is cleared for drawing, then setting carry causes drawing in foreground color, and clearing it draws in the background color.
r3 - x1: x coordinate of pixel (0-319)
r11L - y1: y coordinate of pixel (0-199)

Return: r3, r11L - unchanged

Destroyed: a, x, y, r5 - r6

Synopsis: Draws a point in foreground or background color, or recovers the pixel value from the background screen depending on the values of the carry and sign flags. If the sign flag is set then the pixel is recovered from background. If it is cleared, then if the carry flag is set then the point is drawn to the foreground color (usually black, pixels set to 1 in screen RAM), else the pixel is drawn in the background color (usually white, pixel reset to 0 in screen RAM).

TestPoint

Function: Returns carry with the value of the indicated pixel.

Pass: a - pattern
 r3 - x coordinate of pixel (0-319)
 r11L - y coordinate of pixel (0-199)

Return: carry flag - set if the bit is set, otherwise cleared
 r3, r11L - unchanged

Destroyed: a, x, y, r5 - r6

Synopsis: TestPoint returns the carry flag with the value of the bit whose coordinates are passed in r3 and r11L.

HorizontalLine

Function: Draw a horizontal line with the given pattern

Accessed: **displayBufferOn**

- bit 7 - write to foreground screen if set
- bit 6 - write to background screen if set

Pass:

- a - pattern byte to use for line.
- r11L - y coordinate of line (0-199)
- r3 - x coordinate of left end of line (0-319)
- r4 - x coordinate of right end of line (0-319)

Return: r11L - unchanged

Destroyed: a, x, y, r5 - r8, r11H

Synopsis: Draws a horizontal line from xpositions in r3 to r4. The pattern byte is stored directly into screen RAM bytes to make the horizontal line. If the left endpoint of the line does not fall on a byte boundry, then the correct number of bits are masked off the pattern byte before it is written to the screen RAM. The same is true of the right side of the line. Two lines drawn next to each other in the same pattern will appear to match up, i.e., the set bits in the pattern bytes in each line will be directly above each other.

VerticalLine

Function: Draws a vertical line given a top, bottom, and xposition coordinate

Accessed: **displayBufferOn**

- bit 7 - write to foreground screen if set
- bit 6 - write to background screen if set

Pass:

- a - pattern byte
- r3L - top endpoint of line (0 - 199)
- r3H - bottom endpoint of line (0-199)
- r4 - x coordinate of line (0-319)

Return: r11L - unchanged

Destroyed: a, x, y, r4L - r8L, r11L

Synopsis: Draws a vertical line in the pattern passed in **a**. The pattern byte is used vertically. If a line is drawn in a dotted pattern next to another line drawn in the same dotted pattern, the dots in the two lines will match up as described above. That is, wherever a pixel in one line is black, a pixel in the other line located on the same scanline will also be black. In the same way than pattern bytes are always stored on byte boundries in horizontal lines, vertical pattern bytes are always aligned to divisions of 8 scanlines.

InvertLine

Function: Inverts the bits in a horizontal line

Accessed: **displayBufferOn**

- bit 7 - write to foreground screen if set
- bit 6 - write to background screen if set

Pass: r3 - x coordinate of left endpoint of line (0-319)
r4 - x coordinate of right endpoint of line (0-319)
r11L - y coordinate of line (0-199)

Return: r3 - unchanged
r4 - unchanged

Destroyed: a, x, y, r5 - r8

Synopsis: Inverts all the bits appearing in a horizontal line between the two endpoints.

ImprintLine **RecoverLine**

Function: Imprints a horizontal line into the background screen.
Recovers a horizontal line from the background screen.

Pass: r3 - x coordinate of left endpoint of line (0-319)
r4 - x coordinate of right endpoint of line (0-319)
r11L - y coordinate of line (0-199)

Return: r11L - unchanged

Destroyed: a, x, y, r5 - r8

Synopsis: **ImprintLine:** Copies the bits which make up a horizontal line from the foreground screen to the background screen.

RecoverLine: Same as ImprintLine except in opposite direction.

DrawLine, i_DrawLine

Function: Draws a line in black or white or recovers it from background, between two points on the screen.

Accessed: **displayBufferOn**

- bit 7 - write to foreground screen if set
- bit 6 - write to background screen if set

Pass: sign flag - set to **recover** bits in line from the background screen
 buffer reset for **drawing**
 carry flag - set for drawing in **foreground color** (bits = 1)
 reset for drawing in **background color** (bits = 0)

- r3 - **x1**: x coordinate of first point (0-319)
- r11L - **y1**: y coordinate of first point (0-199)
- r4 - **x2**: x coordinate of second point (0-319)
- r11H - **y2**: y coordinate of second point (0-199)

Inline Pass: data appears immediately after the jsr

- .word - **x1**: x coordinate of first point (0-319)
- .byte - **y1**: y coordinate of first point (0-199)
- .word - **x2**: x coordinate of second point (0-319)
- .byte - **y2**: y coordinate of second point (0-199) word
- .byte - **draw/recover flag** - bit 7 set for drawing in foreground, reset for drawing in background (carry bit above), bit 6 for recover (sign bit)

Return: nothing

Destroyed: a, x, y, r3 - r13

Synopsis: Draws a line between two points on the GEOS screen using the BREESENHAM algorithm (see Fundamentals of Interactive Computer Graphics by J. D. Foley and A. Van Dam, page 435). See this reference to understand which pixels DrawLine decides to draw.

A combination of line drawing features, recovering a line, drawing in the background color (setting bits to 0) and drawing in the foreground color (setting bits to 1) are all provided in DrawLine. The value of the sign

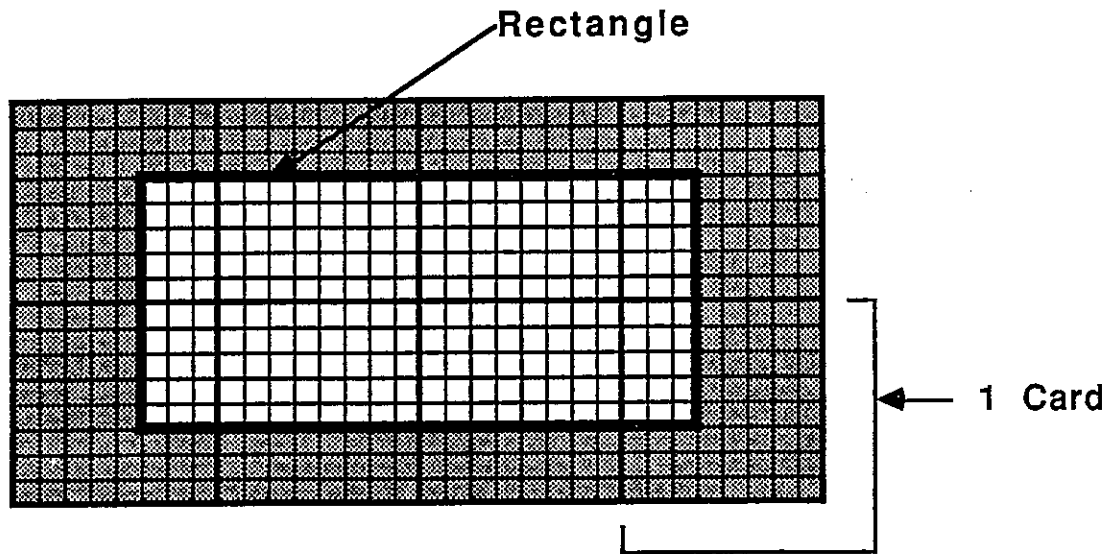
and carry flags (the N and C bits, respectively, of the processor status flag) determine what action is taken. If the carry flag is set then the line is drawn in the foreground color, if it is reset then the line is drawn in the background color. If the sign bit is set then the carry flag is ignored and bits which make up the line are recovered from the background screen.

Drawing Filled Regions

An approach similar to the one used to draw horizontal and vertical lines is used for filling regions with a pattern. In this case the pattern is an 8 x 8 card instead of a byte. The pattern is always aligned to card boundaries when drawn to the screen. 8 x 8 cards are the natural choice for filling in regions of the screen because hi-res bitmap mode is set up like a card oriented display. (See Commodore 64 Programmer's Reference Guide page 121 for a description of graphics modes on the c64.) We've already used two graphics routines in the sample application, `SetPattern` and `i_Rectangle`. `SetPattern` sets the system pattern card. Inside GEOS, an 8 pixel by 8 pixel block of data is set aside (8 bytes) for use as a system pattern. `SetPattern` loads this card with one of several stipple patterns. Each pattern is given a number. These patterns appear at the end of this chapter.

Drawing routines such as `i_Rectangle` use the system pattern when filling areas of the screen. In the case of the sample application, these routines were called to clear the screen. GEOS handles drawing filled regions similarly to horizontal and vertical lines. Regions filled with the same pattern, and placed right next to each other or overlapping will 'line up' just as lines will. The effect is like having a patterned table cloth covered by a white piece of paper. Drawing an object in a pattern is like cutting a hole in the white piece of paper to reveal the pattern underneath. If one draws an object which partially overlaps another, the pattern in the first will match up unbroken against the second.

The diagram below represents a small rectangle which does not conform to card boundaries being drawn to the screen. The grey areas represent bits which are masked so that bits in the pattern byte do not extrude outside the rectangle area which is shown as white in the center.



Grey Area to be Masked When Writing Pattern Card

The following graphics routines are for drawing objects in patterns.

SetPattern

Function: Sets the system pattern card. In the case of pattern bytes,

Pass: a - the pattern number to set (0 - 33)

Return: currentPattern - set to contain the pattern number passed in a.

Destroyed: a

Synopsis: Sets the system pattern to one of the 34 pattern cards defined in the Appendix. The current system pattern number is kept in **currentPattern**. All drawing routines that fill regions such as Rectangle draw with the system pattern.

Rectangle, i_Rectangle

Function: Draws a rectangle in the current pattern to the given coordinates.

Accessed: **displayBufferOn**
bit 7 - write to foreground screen if set
bit 6 - write to background screen if set

Pass: r2L - **top** of rectangle in scanlines (0-199)
r2H - **bottom** of rectangle in scanlines (0-199)
r3 - **left** side of rectangle in pixel positions (0-319)
r4 - **right** side of rectangle in pixel positions (0-319)

Inline Pass: data appears immediately after the jsr
.byte **top** side of rectangle
.byte **bottom** side of rectangle
.word **left** side of rectangle
.word **right** side of rectangle

Return: r11L unchanged

Destroyed: a, x, y, r5 - r8, r11

Synopsis: Draws a rectangle at the given coordinates which is filled with the current pattern. This pattern can be changed by calling SetPattern. The current pattern is an 8 x 8 pixel card which is always drawn to the screen on card boundaries. When the rectangle's boundaries do not fall exactly on card boundaries, the cards along up the top, bottom and sides of the square are filled with the pattern byte and then any part of a card which lies outside the rectangle is masked out. See the discussion at the beginning of this section for more details.

To frame a filled rectangle, call Rectangle **before** calling Frame-Rectangle or else the call to Rectangle will erase the frame.

FrameRectangle, i_FrameRectangle

Function: Draws a line framing a rectangle using the parameter pattern byte. To draw a filled rectangle use **Rectangle**.

Accessed: **displayBufferOn**
bit 7 - write to foreground screen if set
bit 6 - write to background screen if set

Pass: **a** - GEOS pattern byte (the same as passed to **HorizontalLine** and **VerticalLine**)
r2L - top of rectangle in scanlines (0-199)
r2H - bottom of rectangle in scanlines (0-199)
r3 - left side of rectangle in pixel positions (0-319)
r4 - right side of rectangle in pixel positions (0-319)

Inline Pass: data appears immediately after the jsr
.byte top side of rectangle
.byte bottom side of rectangle
.word left side of rectangle
.word right side of rectangle
.byte pattern byte

Return: **r2L** - **r3H** unchanged

Destroyed: **a**, **x**, **y**, **r5** - **r9**, **r11**

Synopsis: Uses the pattern byte as passed in **a** to draw a one pixel wide outline of a rectangle. The pattern byte is turned on its side and used vertically when drawing the left and right sides of the rectangle. As with the other drawing routines, the pattern byte is always written on byte boundaries when drawing horizontally and aligned vertically so that bit 0 of the pattern byte always appears on a scanline evenly divisible by 8. See the discussion at the beginning of this chapter for more details.

To frame a filled rectangle, call **Rectangle** before calling **FrameRectangle** or else the call to **Rectangle** will erase the frame. **FrameRectangle** draws an empty rectangle (i.e., a single pixel wide

line in the foreground color) around a rectangle. To draw a line around a solid filled in rectangle drawn with `Rectangle` or `i_Rectangle`, one must call `Rectangle` before calling `FrameRectangle`. If this order is reversed, the call to `Rectangle` with the same rectangle dimension as `FrameRectangle` will cause `Rectangle` to overwrite the line drawn by `FrameRectangle` since all dimensions in GEOS are inclusive.

InvertRectangle

Function: Inverts a rectangle.

Accessed: **displayBufferOn**
bit 7 - write to foreground screen if set
bit 6 - write to background screen if set

Pass: r2L - **top** coordinate of rectangle in scanlines (0-199)
r2H - **bottom** coordinate of rectangle in scanlines (0-199)
r3 - **left** side coordinate of rectangle in pixels (0-319)
r4 - **right** side coordinate of rectangle in pixels (0-319)

Return: r2, r3 unchanged

Destroyed: a, x, y, r5- r8

Synopsis: The pixels at or contained within the rectangle defined by the given coordinates, are inverted. All 1's go to 0's and vice versa. This has the effect of changing foreground bits to the background color and background colored bits to the foreground color.

RecoverRectangle, i_RecoverRectangle

Function: Recovers a rectangle from the background screen.

Accessed: `displayBufferOn` - ignored

Pass: `r2L` - **top** of rectangle in scanlines (0-199)
`r2H` - **bottom** of rectangle in scanlines (0-199)
`r3` - **left** side of rectangle in pixel positions (0-319)
`r4` - **right** side of rectangle in pixel positions (0-319)

Inline Pass: data appears immediately after the `jsr`
 `.byte` **top** side of rectangle
 `.byte` **bottom** side of rectangle
 `.word` **left** side of rectangle
 `.word` **right** side of rectangle

Return: `r2L` - `r3H` unchanged

Destroyed: `a`, `x`, `y`, `r5` - `r8`, `r11`

Synopsis: The pixels at or within the given rectangle coordinates are copied from the background screen to the same rectangle on the foreground screen. The previous contents of the foreground screen area are lost. Note that drawing to the background screen should have been enabled previous to this call so that there is something on the background screen to copy forward.

ImprintRectangle, i_ImprintRectangle

Function: Copies the bits within a rectangle from the foreground screen to the background screen buffer. This is the opposite of RecoverRectangle.

Accessed: displayBufferOn - ignored

Pass: r2L - **top** of rectangle in scanlines (0-199)
r3H - **bottom** of rectangle in scanlines (0-199)
r3 - **left** side of rectangle in pixel positions (0-319)
r4 - **right** side of rectangle in pixel positions (0-319)

Inline Pass: data appears immediately after the jsr
.byte **top** side of rectangle
.byte **bottom** side of rectangle
.word **left** side of rectangle
.word **right** side of rectangle

Return: r11L unchanged

Destroyed: a, x, y, r5 - r8, r11L

Synopsis: ImprintRectangle takes a rectangular region and imprints it into the background screen buffer at the same place it appears in the foreground screen buffer. It does this by copying the bits contained within the rectangle defined by r2 - r4 to the background screen buffer. A subsequent call to RecoverRectangle with the same parameters will restore the rectangle to the foreground screen.

Bit-Mapped Graphics

When an object or design is too complicated to draw with rectangles, lines, and pattern filled objects, the best way to store it is to just store the bit-map. The problem with bit-maps is that they take up a lot of memory space. Storing bit-maps, therefore, usually goes hand in hand with some type of compaction scheme. In general, bit-mapped data can benefit greatly by some method of run-length encoding. Run-length encoding treats the bit-map data as one long linear string of bytes. Where several identical data bytes appear in series, the same data can be represented more compactly by a byte which contains a count followed by the data byte to be repeated. Compacting data has been the subject of many a scholarly paper, and there are many approaches which can be taken. GEOS bit-maps support three formats for storing the data. GEOS bit-maps are called Bit-Mapped Objects. The formats can be switched at will within the bit-mapped object. This allows the compaction program to tailor the technique being used to compact the data to the character of the data being compacted. Two of the formats employ methods of compaction, while the third indicates a number of unique data bytes to follow with each one appearing once, a strict bit-map.

The first byte in a bit-mapped object is referred to as the COUNT byte. Encoded into this byte is the format and number of data bytes in that format which follow. Together the COUNT byte and the following bit-mapped data make up a COUNT/Bit-map pair. Several COUNT/Bit-map pairs make up a bit-mapped object.

The Compaction Formats

The compaction schemes compact and uncompact data bytes in SCANLINES — horizontally accross the screen — not as byte arranged in cards as they appear in c64 memory.

When uncompactd, any arbitrary byte N appears on the same scanline and immediately to the left of byte N+1 (unless N is the last byte on the line and N+1 the first byte on the next line). This, of course, is totally different from the way graphics data is normally stored in the c64 memory. The reason for this reorganization is that bit-mapped data compacts much better horizontally. After the data in each COUNT/Bit-map pair is uncompactd, it is reordered to be placed correctly within screen RAM: the second byte to be uncompactd, for example, is placed eight memory locations after the first so that it appears

on the same scanline. (This makes compaction slower, but since the disk is so much slower than anything else, saving disk space and seek time by decreasing the number of blocks to write turns out to be a worthwhile tradeoff.) The methods of compaction are discussed below.

Each COUNT/Bit-map pair begins with a COUNT byte. The value of this byte determines the format of the following data. If COUNT is within 0 to 127 then the first format is indicated. 128 to 220 indicates the second format, 221 to 255 is the third. The byte following COUNT in the first format is repeated COUNT times. Thus if COUNT were 100 and the following byte were 0, then this would uncompact to 100 consecutive 0's in the bitmap. If COUNT takes a value within 128 to 220, then the following (COUNT-128) bytes are straight bitmap, each used once. Thus if COUNT were equal to $128 + 35 = 163$, then this would indicate that COUNT is followed by 35 unique bytes. If COUNT takes on a value within 221 to 255, then the following data is in BIGCOUNT format. BIGCOUNT is an interesting animal and deserves greater explanation.

BIGCOUNT was invented as a way of repeating a pattern which takes up several bytes. That is, suppose you had a 4 byte repeating pattern:

xxxy xxxy xxxy xxxy.

To display this in BIGCOUNT format you must first describe the 4 byte pattern in one of the first two modes. xxxy can be described in the second format as (132 -128) xxxy, which says use the next 4 bytes once each. You can also describe this pattern as 3x 1y, which is two entries in the first format telling GEOS to use a three times and b once.

Now that the pattern to repeat is described you need to tell GEOS how long the pattern description is and how many times to repeat it. The first byte is the number of bytes in the pattern, and is presented as (COUNT - 220). This is the COUNT byte. The byte immediately following the COUNT byte is referred to as the BIGCOUNT byte. It contains the number of times to repeat the multibyte pattern. The above pattern can thus be represented as the following string:

224 4 3x 1y

This saves ten bytes out of a 16 byte pattern. Below is a table that summarizes the three formats.

COUNT	FORMAT	DESCRIPTION
00 - 127	COUNT DATABYTE	Use next byte COUNT times
128 - 220	COUNT DATABYTE, ...	Use next (COUNT-128) bytes once each.
221 - 255	COUNT BIGCOUNT PATTERN	COUNT - 220 = number of bytes in pattern to follow. Doesn't count BIGCOUNT. BIGCOUNT = number of times to repeat the PATTERN. PATTERN describes a pattern using the first two formats.

To summarize, the bit-mapped object is a collection of COUNT/Bit-map pairs in different compaction formats. A COUNT/Bit-map pair consists of a format byte followed by a series of data bytes in the indicated compaction format. After being uncompact, the data is reordered from scanlines to cards.

BitmapUp, i_BitmapUp

Function: Put up a rectangular bit-mapped object from a compacted bit-map.

Accessed: **displayBufferOn**
bit 7 - write to foreground screen if set
bit 6 - write to background screen if set

Pass: r0 - pointer to the bit mapped data in COUNT/Bitmap format
r1L - **x position** in bytes of left side of the bit-map (0-39)
r1H - **y position** in scanlines for top of bit-map (0-199)
r2L - **width** in bytes of the bit-map (0-39)
r2H - **height** in pixels of the bit-map (0-199)

Inline Pass: data appears immediatly after the jsr
.word - pointer to the bit mapped data in COUNT/Bit-map format
.byte - **x position** in bytes of left side of the bit-map (0-39)
.byte - **y position** in scanlines for top of bit-map (0-199)
.byte - **width** in bytes of the bit-map (0-39)
.byte - **height** in pixels of the bit-map (0-199)

Return: r11L - unchanged

Destroyed: a, x, y, r0 - r9L

Synopsis: BitmapUp displays a compacted GEOS bit-mapped object on the screen. BitmapUp uncompresses and places the bit-map according to the position and dimension information are passed by the caller. A discussion of the compaction formats used in GEOS are discussed above. BitmapUp does not check to see if the position or dimensions passed are legal. To force a bit-map within an area, use BitmapClip below.

It is sometimes useful to display only a portion of a bit-mapped object. The routine below, `BitmapClip`, allows the programmer to constrain a bit-mapped object to a specified window on the screen. Only the portion of the bit-map that appears inside the window will be drawn. The remainder is "clipped" off.

There is also the ability to specify what part of a large bit-map to display within the window defined on screen. The bit-map may be shifted left or right under the window with only the part of the bit-map that appears under the window area being drawn. To clarify this a bit, most programmers will figure out the coordinates of the window on the screen in which they want to display the bit-map. These are passed in `r1` and `r2` as described below. Next in `r11L` and `r11H` the programmer specifies the number of bytes to skip over before beginning to print the graphics within the window. A value of 0 in `r11L` means that the leftside of the graphic will appear at the left side of the window. A value of 10 in `r11L` means ten bytes should be skipped before beginning to print within the window.

If the graphic is wider than the window then after the part of the graphic that fits in the window has been printed, the remaining bytes on that line should be skipped. This value is stored in `r11H`. The width of the bit-mapped object is then divided up into the number of bytes to skip before writing to the screen, stored in `r11L`, the width of the window in bytes, stored in `r2L`, and the remaining bytes stored in `r11H`.

Vertical clipping is handled in a similar way. `r12` is a full word variable that contains the number of scanlines from the top of the bit-mapped object to skip before beginning to display the bit-map within the window. This is how the top and bottom are clipped. Increasing the number passed in `r12` scrolls the bit-map up under the window. `BitmapClip` appears below.

BitmapClip

Function: Display, and clip to fit if necessary, a subset of a bit-mapped object within an indicated window on the screen.

Accessed: **displayBufferOn**

- bit 7 - write to foreground screen if set
- bit 6 - write to background screen if set

Pass:

- r0 - pointer to the bit-mapped object
- r1L - **leftside** of window in bytes to display the bit-map (0-39)
- r1H - **top** of window in pixels to display the bit-map (0-199)
- r2L - **width** of window in bytes to display the bit-map (0-39)
- r2H - **height** of window in pixels to display the bit-map (0-199)
- r11L - number of bytes from the beginning of each pixel wide row of the bit-map image to skip before printing within the window on the screen.
- r11H - number of bytes remaining in each row after printing the portion of the bit-map which fits in the window specified by r0 - r2.
- r12 - A word value specifying a number of scanlines to skip before displaying within the window. This is how top and bottom clipping is achieved.

Return: nothing

Destroyed: a, x, y, r0 - r12

Synopsis: BitmapClip is used to print a portion of a bit-mapped image at an indicated position on the screen. This position is usually an application's work space area. R1L and r1H together with r2L and r2H, define a window in the screen where a part of the bit-mapped object will be displayed. r1L and r1H contain the position of the upper left corner of the window, while r2L and r2H define the window's dimensions. r0 contains the address of the beginning of the bit-mapped object..

Since the width of the bit-mapped object being displayed may be larger than the window, the programmer must indicate what part of the bit-mapped object is to be displayed within the window. This information is passed in R11L and r12. As BitmapClip uncompresses each scanline, it looks at R11L which contains the number of bytes to skip in each line of

the bit-mapped object before beginning to display it in the window. R11H contains the number of bytes remaining in the row of the bit-mapped object after printing the bytes fitting in the window. Note that the width of the bit-mapped object = r2L (window width) + r11L(skip before) + r11H (skip after). R12 contains the number of complete lines to skip from the top of the bit-mapped object before beginning to print lines within the window.

Sometimes the application programmer will need to display a clipped bit-map but cannot afford to store the entire bit-mapped object in memory. Bit-maps can get very big. BitOtherClip allows the caller to specify an input routine that returns the next uncompact byte in r0. BitOtherClip will call the input routines, usually just ReadByte, until it has assembled one COUNT/Bit-map pair. It then uncompresses the bit-map and displays it in a window in the screen whose dimensions have been passed by the user.

BitOtherClip

Function: Allows programmer to specify an input routine to be used with BitmapClip.

Accessed: **displayBufferOn**
bit 7 - write to foreground screen if set
bit 6 - write to background screen if set

Pass:

- r0 - pointer to a 134 byte buffer area.
- r1L - **leftside** of window in bytes to display the bit-map (0-39)
- r1H - **top** of window to display the bit-map (0-199)
- r2L - **width** in bytes of the window on screen to display the bit-map (0-39)
- r2H - **height** in pixels of the window to display the bit-map(0-199)
- r11L - number of bytes from the beginning of each pixel wide row of the bit-map image to skip before printing within the window on the screen.
- r11H - number of bytes remaining in each row after printing the portion of the bit-map which fits in the window specified by r0 - r2.
- r12 - a word value specifying a number of scanlines to skip before displaying within the window. This is how top and bottom clipping is achieved.
- r13 - address of the input routine, returns next byte from compacted bit-map in a.
- r14 - address of sync routine. Due to improvements in BitmapClip this routine need consist only of reloading r0 with the address of the 134 byte buffer.

Return: nothing

Destroyed: a, x, y, r0 - r14

Synopsis: Sometimes the application programmer will need to display a clipped bit-map but cannot afford to store the entire bit-mapped object in memory. Bit-maps get very big. BitOtherClip allow the caller to specify an input routine that returns the next compacted byte (in BitmapUp form) in r0. This routine should not clobber r0 - r13. Usually this routine returns a byte at a time (buffered) from a bit-mapped object stored on disk. Often this means just saving registers,

calling ReadByte (see the file section for details on ReadByte) and returning the input byte in a.

BitOtherClip calls the routine in r13 until it has enough bytes to form one COUNT/Bit-map pair. BitOtherClip stores the bytes in the buffer pointed to by r0. It then uncompresses the COUNT/Bit-map pair and writes it to the screen. When this is finished the routine in r14 is called. This routine is no longer particularly useful and should just reload r0 with the address of the 134 byte buffer.

When GEOS was being developed, it was found that the graphics routines were most useful in setting up the initial appearance of a screen. Used for this purpose, many routines would often be called in a row. A shorthand way to do this was, therefore, developed. It is the GraphicsString routine. It allows the programmer to create a string of calls to graphics routines. Several of the routines are given numbers and the graphics string consists of these numbers followed by their arguments. In this way all the bytes for loading parameters into pseudoregisters and for the jsr instruction are saved.

GraphicsString has several interesting features worthy of note. First a current position is kept as the pen position. Most graphics string commands use the pen position to draw from. Thus LINETO draws a line from the current pen position to the x, y position immediately following in the string. This new position then becomes the position of the pen.

All lines and rectangle borders drawn in a graphics string are draw solid. There is no ability to draw dotted lines using a pattern byte. GraphicsString appears below.

GraphicsString, i_GraphicsString

Function: Executes graphics drawing commands embedded in a string format.

Accessed: **displayBufferOn**

- bit 7 - write to foreground screen if set
- bit 6 - write to background screen if set

Pass: r0 - pointer to the beginning of the graphics string

Inline Pass: data appears immediately after the jsr
the string is stored immediately after the jsr

Return: nothing

Destroyed: a, x, y, r0 - r13

Synopsis: GraphicsString executes a string made up of graphics commands. There are nine graphics commands numbered from 0 to 8. Each command's number is followed in the graphics string by any x, y position data needed. X coordinates are two bytes long, 0-319; y coordinates are one byte long, 0-199. The commands are:

Command	No.	Data	Description
NULL	0	-	End of graphics string.
MOVEPENTO	1	.word <i>x</i> , .byte <i>y</i>	Move the pen drawing position to the absolute coordinates (<i>x</i> , <i>y</i>).
LINETO	2	.word <i>x</i> , .byte <i>y</i>	Draw a line from the current pen position to (<i>x</i> , <i>y</i>), which becomes new drawing position.
RECTANGLETO	3	.word <i>x</i> , .byte <i>y</i>	Draw a rectangle from the current drawing position to (<i>x</i> , <i>y</i>), which becomes the new drawing position.
	4	unused	
NEWPATTERN	5	.byte <i>patternNo.</i>	Load system pattern with new pattern.
ESC_PUTSTRING	6		Switch to interpreting the remainder of the string as putstring commands.
FRAME_RECTO	7	.word <i>x</i> , .byte <i>y</i>	Frame a rectangle using the pattern byte. Start at the current drawing position to (<i>x</i> , <i>y</i>), which becomes the new drawing position.

Example: Draw a simple rectangle with pattern 0 with upper left corner at (leftside,top), and lower right at (rightside,bottom).

```

jsr    i_GraphicsString
.byte  NEWPATTERN,    0
.byte  MOVEPENTO,     [leftside, ]leftside,  top
.byte  RECTANGLETO,   [rightside, ]rightside, bottom
.byte  NULL

```

GetScanLine

Function: Returns the address of the beginning of a scanline.

Pass: x - scanline number

Return: r5 - address of first byte in scanline screen RAM
r6 - address of first byte in scanline in background buffer

Destroyed: a

Synopsis: GetScanLine returns the address of the first byte in the scanline whose number is passed in x. r5 returns the address of the first byte in the scanline in foreground screen RAM, while r6 contains the address of the first byte in the scanline in the background buffer.

Text in GEOS

Built-in text handling with multiple fonts, styles, and sizes is one of the greatest advantages of GEOS. All of GEOS's text and character features are accessible through applications. As it happens, some applications may not need to use all the features while others such as desktop publishing programs will use everything GEOS has to offer and may even add more. To satisfy both those who demand simplicity and those who demand flexibility, two levels of routines are supported. At the most simple level GEOS supports several sentence level input and output commands. At this level, GEOS handles all the character spacing and style changes like boldface and italics for an entire sentence. Both inputting sentences from the user and echoing them back as well as writing sentences to the screen are provided. Changing fonts, or font sizes within the sentence, however, are not directly supported because of memory management issues involved in manipulating fonts: fonts can be rather large.

The routines to construct a font management system are provided and are easy to use. With a little more work an application may swap fonts, change sizes, and styles at will. To do this GEOS provides access to text routines at the character level. To change fonts in

The routines to construct a font management system are provided and are easy to use. With a little more work an application may swap fonts, change sizes, and styles at will. To do this GEOS provides access to text routines at the character level. To change fonts in mid-line requires that the application routine set aside extra memory space for the font data, read the font in, and call the character drawing routine directly.

In the first part of this chapter we discuss quick and simple text usage. The second section discusses using the complete font usage.

Simple String Manipulation

The simplest way to support text I/O in GEOS is with the routines `PutString` and `GetString`. `PutString` will print a text string specified by the programmer. `GetString` will wait for the user to type in a string and return that string in a buffer. A common way to use these two routines is to have `PutString` to print a sentence, prompting the user for input, and then call `GetString` to retrieve that input. First let's see how `PutString` is used.

Most text strings in GEOS are null terminated. This means that the end of the string is signalled by a zero. The following is an example of a null terminated string:

`TextLabel:`

`.byte "This is a null terminated string.",0`

Our assembler recognizes characters inside double quotes should be stored as ASCII. Thus the ASCII for the message `This is a null terminated string.` is stored in consecutive bytes. A zero is stored in the byte after the period indicating the end of the string. `PutString` will print a string such as this at a location on the screen specified by the programmer. A call to `PutString` to print the above string looks like the following:

<code>LoadW r0,TextLabel</code>	<code>;r0 points to the text string</code>
<code>LoadB r1H,YPOSCONSTANT</code>	<code>;yposition to put string. Possible range 0-199</code>
<code>LoadW r11,XPOSCONSTANT</code>	<code>;xposition for string. Possible range 0-319</code>
<code>jsr PutString</code>	

where `YPOSCONSTANT` and `XPOSCONSTANT` are the coordinates at which to place the left side and baseline of the first character (this position is easiest to think of as the lower left corner of the first character). There is also an inline form of `PutString`, `i_PutString`. A call to it

looks like this:

```

jsr      i_PutString          ;call the routine
.word    XPOSCONSTANT        ;the inline xposition, Possible range 0-319
.byte    YPOSCONSTANT        ;the inline yposition, Possible range 0-199
                                ;the string to print:
.byte    "This is a null terminated string.",0
...                                           ;code resumes here

```

Fault Vectors

What if something goes wrong? Suppose the string you ask PutString to print goes off the right side of the screen. As soon as this happens, GEOS will look for an address stored in StringFaultVector. GEOS initializes this vector to zero. If it is still zero when PutString encounters the fault, then PutString will simply not print the character. If the application has stored the address of a routine in StringFaultVector, then GEOS will call that routine. GEOS passes the StringFaultVector routine the ASCII value of the offending character in a, the x position to print that character, is passed in r1l and the y position in r1H.

The actual fault positions that PutString uses are stored in the variables leftMargin, rightMargin. The application supplied StringFault dispatch routine may check the x and y position, see which margin was violated and reposition the string. Usually the StringFault dispatch routine will want to print the out-of-bounds character itself. This can be done with Putchar which we describe below, and therefore this discussion is best postponed till then.

Embedded Style Changes

PutString will also handle embedded style change characters. These are special ASCII values that normally represent unprintable characters, such as the control characters. The first printable character in ASCII is 32 for the space. GEOS uses ASCII values less than 32 to signal changes in printing such as turning on boldface or italics. We refer to this as a style escape because it causes PutString to escape from normal processing for a moment to switch styles. Beginning with the next character, PutString will output in boldface. What happens is something like this: PutString will be reading and printing characters to the screen when it encounters, for example, a 24 (decimal) in the input string. When embedded in an

input string, 24 signals a change to boldface. We refer to this as a style escape because it causes PutString to escape from normal processing for a moment to switch styles. PutString then prints the characters following the embedded 24 in boldface.

When it encounters a style escape byte like this, PutString will store the new style value in the GEOS variable `currentMode`. `currentMode` contains bits for each possible style. When an escape is encountered the bit for the new style is set in `currentMode`. The present mode is not unset. Thus if the present mode is boldface and the escape for italics is encountered, the following characters will be printed in boldface italics. The bold bit remains set and the italic bit is added to it. The only exception to this is the escape for plain text. In this case all the bits in `currentMode` are cleared returning printing to plain unembellished text.

When the input string terminates, `currentMode` retains its value. The next call to PutString, or any of the other character printing routines explained later, will continue printing in that style. To guarantee printing in a particular style, an input string passed to PutString should begin with an escape to plain text, followed by escapes for the style it wants for printing.

Font Change Escape

PutString may also encounter an embedded font change escape in an input string. This consists of the value 23 (decimal) followed by a word indicating the font ID number and one of the style bytes. A style byte always follows the font ID because it makes supporting multiple fonts easier if the programmer knows what style is active whenever a font escape is encountered. As mentioned above, PutString doesn't have the capability to deal with fonts and so ignores the font change. PutString also ignores the style change. (It would have been possible to have it honor the style byte only.)

Position Escapes

There are four motion escapes that reset the position of the cursor on the page, 1 forces the position of the mouse cursor to the upper left corner of the screen. The x, y position is set to 0, 0. UPLINE will move the cursor up the height of the current font being used. GOTOX and GOTOY will place the cursor at a specified position. The word following GOTOX is used as the xposition. Legal values for it are in the range 0 to 319. For GOTOY the byte following the GOTOY is used as the yposition. Its legal values are 0 to 199.

The possible style, font, and position escapes appear below.

Char	ASCII	Function
NULL	0	Terminates string
BACKSPACE	8	Erase the previous char whose width is stored in lastWidth. Without support from the application to reload lastWidth, this only works once
FORWARDSPACE	9	Move right the width of space char in current font
LF (line feed)	10	Move down one line (value of currentHeight)
HOME	11	Move to upper left corner of screen
UPLINE	12	Move up one line (value of currentHeight)
CR (carriage rtn)	13	Move to beginning of next line: x position is set to leftMargin, and LF is automatically executed.
UNDERLINEON	14	Turn on underlining
UNDERLINEOFF	15	Turn off underlining
REVERSEON	18	Turn on reverse video
REVERSEOFF	19	Turn off reverse video
GOTOX	20	Use next word as new xposition to store in r11
GOTOY	21	Use the next byte as new yposition to store in r1H
GOTOXY	22	Use next three bytes as x, y position (x first)
NEWCARDSET	23	Ignore the following font ID word and style byte
BOLDON	24	Turn on boldface
ITALICON	25	Turn on italics
PLAINTEXT	26	Return to plain text

PutString

Function: Draws a character string on screen

Accessed: **displayBufferOn**

bit 7 - write to foreground screen if set

bit 6 - write to background screen if set

leftMargin, rightMargin, windowTop, windowBottom - if the string to be printed goes outside of these boundaries, then the routine in **StringFaultVector** is called. If this is zero any characters outside these margins are not printed

Pass: r0 - address of null terminated text string
r1H - **y position** on screen to display text (0-199)
r11 - **x position** on screen to display text (0-319)

Inline Pass: data appears immediately after the jsr
.word - **x position** on screen to display text (0-319)
.byte - **y position** on screen to display text (0-199)
<the text string>

Return: r11 - **x position** for next character to be drawn
r1H - **y position** for next character to be drawn

Destroyed: a, x, y, r0L, r2 - r10L

Synopsis: PutString prints the null terminated text string pointed to by (r0) at the given x, y position. PutString will draw text in the currently active font. To change fonts call LoadCharSet or UseCharacterSet. All special characters accepted by Putchar are accepted by PutString plus several multibyte command sequences. Each command sequence is made up of a nonprintable ASCII char followed by any position bytes needed. Text support in GEOS is discussed in detail in the "Font and Text" section of the Commodore 64 Programmer's Reference Guide and also in GEOS Technical Paper 2: The Photo and Text Scrap.

PutString will skip over NEWCARDSET command strings encountered in the text string. The reason for this is PutString does not have the facility to dynamically manage font information. The following table contains the available embedded char commands.

PutDecimal

Function: Draws a 16 bit number in decimal on the screen.

Accessed: **displayBufferOn**

bit 7 - write to foreground screen if set

bit 6 - write to background screen if set

leftMargin, rightMargin, windowTop, windowBottom - if the string to be printed goes outside of these boundaries, then the routine in **StringFaultVector** is called. If this is zero any characters outside these margins are not printed

Pass: a - Format: bit 7: 1 for left justify
0 for right justify
bit 6: 1 for suppress leading 0's
0 for print leading 0's
bits 0 - 5: field width when using right justify format
r0 - 16 bit number to print
r1H - **y position** on screen to display number (0-199)
r11 - **x position** on screen to display number (0-319)

Return: r11 - **x position** for next character to be drawn
r1H - **y position** for next character to be drawn

Destroyed: a, x, y, r0, r2 - r10, r12, r13

Synopsis: PutDecimal converts a 16-bit binary number to ASCII and sends the resulting characters to Putchar. Putchar will print one character at a time and is described in detail below. If right justify format is used, the width of the field must be passed in bits 0 - 5 of the accumulator. The right most pixel position can then be calculated by adding this width to the x position for the number as passed in r11.

String Input

Simple string input is done with `GetString`. Often `PutString` is called before `GetString` to prompt the user for the input. `GetString` is then called to echo characters to the screen and save the input string in a buffer. `PutString` returns the x, y position just past the last character printed. This is the position the next character would have been printed, had the string been one character longer. Add a couple of pixels of space to this position, and you have a good position to echo the user entered input string.

`GetString` also needs a few other bits of information. First, it needs the address of a buffer to stick the entered string, and the length of the buffer. As characters are entered they will be stored in the buffer. When the user hits return a zero is stuck at the end of the string to null terminate. `GetString` also requires a maximum number of characters to accept. If the user types too many characters before pressing return a string fault will occur. The programmer has the option of passing `GetString` the address of a routine to call if this happens, or to use the default. This default null-terminates the input string where the fault occurred, usually the last char entered before overflowing the input buffer.

GetString

Function: Obtains an input string from the GEOS user.

Accessed: **displayBufferOn**
bit 7 - write to foreground screen if set
bit 6 - write to background screen if set

leftMargin, rightMargin, windowTop, windowBottom - if the string to be printed goes outside leftMargin or rightMargin, then the routine in **StringFaultVector** is called. If SFV is zero any characters outside these margins are not printed. Any character or part of a character appearing above windowTop or below windowBottom will be clipped. For example, if a line of characters is printed too close to windowTop, the tops of the characters on that line will be clipped off.

Pass: **keyVector** - address of routine to call when user input has been accepted.
r0 - address for getString to put user entered string.
r1L - Reserved for bit **Flags**. If bit 7 is set, use the user supplied string fault vector pointed to by r4.
r1H - **y position** in scanlines to begin echoing input characters to the screen (0-199).
r2L - **max chars** to accept for string.
r11 - **x position** in pixels to begin echoing input (0-319)
r4 - (optional) user supplied max char fault vector

Return: nothing

Destroyed: a, x, y, r0 - r13

Synopsis: GetString provides a convenient way for applications to prompt the user for character input such as filenames, etc. getString will gather all user input up to a carriage return and place it in the buffer pointed to by r0. When the user has typed return, the routine whose address is passed in keyVector is called. This routine then processes the input string which getString has placed in the buffer pointed to by r, and has null terminated.

The input string is echoed to the screen as the user types. Two parameters are passed with the screen x, y position at which to begin echoing chars. Also passed is the maximum number of characters expected in the input string. The address of a routine to be called if the user types in more characters than the keyVector knows how to handle is also passed. This is called the **max char fault vector**. This routine is only used if bit 7 of the Flags byte parameter is set. Otherwise the parameter passed in r4 is ignored, (or in the case of an inline call, the final word should be omitted), and the default GEOS max char fault vector is used. This default null-terminates the input string where the fault occurred, usually the last char entered before overflowing the input buffer.

If the user manages to type off the end of the screen, specifically past right Margin, GetString will stop echoing characters although it will still enter the characters into the buffer.

6.

Character Level Routines

For many applications, GetString and PutString provide adequate text support. These two routines, however, have been pared down to provide only the lowest level of string support that is still generally useful. Anything more than this would unnecessarily rob applications that didn't need elaborate text support of precious code space. To provide more complicated text support for your application you will need to use the character level routines.

Character routines support:

1. reading and writing characters at a specified coordinate,
2. placing a text prompt (a vertical bar),
3. swapping fonts, and
4. getting the width and height of a character in the current Font and point size.

With these few routines it is possible to build a sophisticated word processor. To show how these routines fit together we can build a simple version of GetString. For want of

With these few routines it is possible to build a sophisticated word processor. To show how these routines fit together we can build a simple version of GetString. For want of a better name, let's call it OurGetString. It will read buffered input from the keyboard, display and update the text prompt position so that it moves ahead of the text, and echo the characters back to the screen. When we get this running we can generalize it by adding support for reading embedded control characters. OurGetString can then be used as the basis for a text editor module that reads from a buffer as well as/instead of from the keyboard.

We begin by looking at keyVector, and keyData. keyVector contains the address of the keyboard dispatch routine. keyData gets the value of the key that was pressed. The keyVector routine gets called every time GEOS detects that a key was hit. Initially keyVector is set to 0 by the GEOS Kernal so all characters typed from the keyboard will be ignored. The application should load keyVector with the address of a routine to handle character input. In the present case this is the address of OurGetString.

When a key is pressed on the keyboard, the Interrupt Level code in GEOS places the ascii value of that key in the variable keyData. Interrupt Level checks this every 60th of a second. During MainLoop, GEOS will check a flag left by Interrupt Level and if it indicates that a key has been pressed, MainLoop will call OurGetString. OurGetString can then get the character value out of keyData.

MainLoop does a little more than this though. If the application is doing alot of processing, then it is possible that the user may have had a chance to enter two or three characters since the last call through keyVector to OurGetString. In this case, GEOS automatically buffers keyboard input. If Interrupt Level finds that another key has been pressed, and keyVector hasn't been serviced, it saves the character in its own internal buffer. The routine GetNextChar can then be call from within the keyboard dispatch routine to retrieve characters stacked up in the input buffer. Each time GetNextChar is called it returns the next character from the input buffer. When there are no more characters to return, GetNextChar returns zero.

When OurGetString is called, we retrieve the first character from keyData. We then call GetNextChar in a loop to return the remaining characters. Each time we get a char-

acter we store it in our own input buffer, `inBuffer`.

As we retrieve the input characters we will want to echo them back. This means calling `Putchar` to print it to the screen. You pass `Putchar` the character to print and an `x` and `y` position on screen to print it at. The position can be any legal position on the screen, 0 to 319 for `x`, and 0 to 199 for `y`. `Putchar` is the same routine used by `GetString` and `PutString`.

It is also possible to use `StringFaultVector` to handle printing off screen, or outside of margins. `StringFaultVector` will get called when `Putchar` tries to print a character outside of the `leftMargin`, `rightMargin`. `Putchar` will also clip any part of a character that appears outside of `windowTop` and `windowBottom`. Clipping means that any part of a character appearing outside the top and bottom margins will not be printed. Therefore on the top and bottom edges of a text window, chopped off characters may appear. This is useful for implementing scrolling where characters may be of different fonts and sizes on the same line.

`StringFaultVector` can be used to scroll a text window left or right or to wrap characters from the right side of the screen to the left. In the first case, if the text window as defined on the screen by `windowTop`, `windowBottom`, `leftMargin` and `rightMargin` is used as a window overlooking a much larger document, then it is natural to want to scroll the document under the window. When a character is entered that lies outside the window, the `StringFaultVector` routine is called and may then erase the text in the window area and redraw it shifted to the left to make room for the new text on the right.

Our `StringFault` dispatch routine will perform a simple character text wrap. Characters typed past the end of the line will be moved to the beginning of the next. It will look at the height of the current line, add that to the vertical position of the text and use the result as the new vertical position. `leftMargin` is used as the new horizontal position. When the `StringFault` dispatch returns, it returns the same as if `Putchar` had returned. Our `GetString` will not know that `StringFault` was ever triggered. All it knows is that it called `Putchar` and a character was printed.

To briefly recap, `OurGetString` will Prompt the user for input, display the text prompt, and get keyboard data from reading `keyData` and calling `GetNextChar`. As the characters are entered they will be echoed via `Putchar` and stored in our own internal buffer. If the end of the line is reached before the user hits return, our `StringFault` dispatch will perform a character wrap.

The routine begins with the call to `PutString` to print the prompt.

```

jsr      i_PutString      ;call the routine
.word    XPOSPROMPT      ;the inline xposition, Possible range 0-319
.byte    YPOSPROMPT      ;the inline yposition, Possible range 0-199
                        ;the string to print:
.byte    "Enter something here: ",0
...                        ;code resumes here

```

Now we should put up the text prompt. To do this we need to set the size and position. For now we will be printing in the standard GEOS character set which is 9 point and so let's choose 12 for the size of the vertical bar. The x, y position for the bar is easiest to find by experiment, trying a value and running the program. For now let's define the constants XPOSPROMPT and YPOSPROMPT and guess at their initial values, later.

```

XPOSPROMPT = some x value in range 0 to 319
YPOSPROMPT = some y value in range 0 to 199

```

Next we call PromptOn in order to turn on the sprite used for the text prompt and position it. The text prompt uses sprite 1.

```

lda      #9                ;pass height of text prompt
jsr      InitTextPrompt    ;init the prompt

LoadW    stringX, XPOSPROMPT ;pass the x and ypos for prompt
LoadB    stringY, XPOSPROMPT
jsr      PromptOn          ;make it visible

```

StringX and stringY are the variables used by PromptOn to hold the x, y position of the prompt. The cursor is now visible. OurGetString will get a character, print it to the screen, and then move the prompt to the right of the character. Luckily Putchar returns r1 and r11 updated to for the width of the char. All we need to do is transfer the updated x-position to stringX. So let's start writing OurGetString.

The first thing to do is make sure we get called. Let's load keyVector with OurGetString's address. While we're at it let's do the same for our string fault vector routine. Add the following line to the prompting code above.

```

LoadW    keyVector, OurGetString ;set up keyboard dispatch
LoadW    StringFaultVector, OurStringFault ;set up keyboard dispatch

```

Let's take a close look at OurGetString. It gets the first character from keyVector, checks for the carriage return the user types to terminate the input string. If the character is not a CR then we echo it with Puchar, and store it in the input Buffer. Next, GetNextChar is called to return any additional chars until it returns zero. As part of echoing each input character, OurGetString will advance the text prompt the width of the character. Since stringX and stringY are used to pass the x, y position for the text prompt to PromptOn, we also use them to hold the position to print the input characters as well. The code is as follows.

OurGetString:

```

ldx    #0                ;used as index into our buffer
lda    keyData            ;get first key

charloop:
cmp     #CR               ;see if user indicates end of string
beq     endString         ;if so go terminate the string

sta     inBuffer,x        ;add to our input buffer
pha     ;save the char
inx     ;point to next open byte in inBuffer

MoveW   stringY,r1H       ;Get pos for char from stringX and
MoveW   stringX,r1l       ;stringY, the pos of the prompt.
pla     ;get the character from stack
jsr     Puchar            ;echo the char to the screen
                        ;Puchar returns new x-y pos
                        ;in r1 and r1l, use for prompt
MoveW   r1l, stringX;    Get x-pos for next char into
                        ;stringX. Only xpos changed.
jsr     PromptOn         ;update the prompt position

jsr     GetNextChar       ;
cmp     #0               ;see if last character
bne     charloop         ;if nonzero then more chars

endString:
lda     #0               ;was zero so exit
sta     inBuffer,x       ;terminate the input string in
                        ;inBuffer
rts
```

We can now input and echo characters to the screen. Eventually though, OurGetString will try to print a character past rightMargin, and OurStringFault will get called. We want it to change the x,y position of the text prompt and the location for drawing upcoming characters to the next line. In order to reset the y-position to the next line, OurStringFault has to know how tall the characters on the present line are. The easiest way to do this is to use the routine, GetRealSize. OurStringFault should save the character passed to it, and call GetRealSize to find out the height of the character. It needs to add this height plus a little more to space the lines apart to the present vertical position in stringY. stringX is set to the left margin and the character is printed.

OurStringFault:

```

        pha                ;save the char passed us
        ldx    currentMode ;style may affect char width

        jsr    GetRealSize ;we want the height
        txa                ;height returned in x
        clc
        adc    stringY     ;add height to stringY
        adc #2             ;add a little line spacing
        sta    stringY     ;new y-position
        LoadW stringX,leftMargin ;print from left margin
        pla                ;restore the char
        jsr    Putchar     ;print the char at begin'n of line
        rts

```

The above two routines provide a simple GetString. The input string is stored in the array inBuffer. It is easy to expand OurGetString to provide the same functionality GetString does: passing the address of a buffer to store the string in, passing the max number of chars, and passing the address of a string fault vector.

Now that we can input and print in plain text, the next step is to add style and font changes to our strings. We will take up there after presenting the routines used above.

GetNextChar

Function: Returns characters from the buffered input queue. When no more characters are available, will return 0.

Pass: nothing

Return: a - character from input buffer or 0 if buffer empty

Destroyed: x

Synopsis: GEOS buffers keyboard input. When Main Loop processing slows down as when an application does extensive screen redrawing, more than one character may be typed before the application gets around to accepting keyboard input again. Each time it is called, GetNextChar will either return the next char from the input buffer in a, or zero if there are no more chars. Typically an application wants to retrieve all the input since the last time it read input. It transfers GEOS's internal input buffer to its own input buffer by calling GetNextChar in a loop until it returns 0.

InitTextPrompt

Function: Sets up the size of an edit cursor, the vertical bar which marks an insertion point in a text string.

Pass: a - the size in pixels to make the edit cursor

Return: nothing

Destroyed: a, x, y

Synopsis: A large vertical bar is often to display the current insertion point when accepting user input as with getString. InitTextPrompt will create this prompt cursor with the size passed in a. Sprite one is used for the cursor.

PromptOn

Function: Displays the Text Cursor at (stringX, stringY)

Pass:
stringX- **x position** in pixels to display Text Cursor (word: 0-319)
stringY- **y position** in scanlines for the Text Cursor (byte: 0-199)

Return: nothing

Destroyed: a, x, r3L, r5L, r6

Synopsis: **PromptOn** turns on the Text Cursor (sprite1) and positions it at stringX, stringY. The Text Cursor should have already been initialized with a call to InitTextPrompt.

PromptOff

Function: Erases the Text Cursor from the screen

Pass: nothing

Return: nothing

Destroyed: a, x, r3L

Synopsis: **PromptOff** is used to disable the text prompt. Use prompt off as follows:

```
php
sei
jsr    PromptOff
LoadB alphaFlag,0
cli
plp
```

contains bits that govern the text prompt. If bit seven is set then there is a text prompt. In this case, bit 6 will tell us the status of the prompt. While the prompt is blinking on and off bit 6 being set indicates that the prompt is visible. When bit 6 is clear then the prompt is invisible. The lowest 6 bits are a counter for switching from on to off and back.

Calling PromptOff will disable the text prompt. Setting alphaFlag to 0 above will keep GEOS from turning the prompt back on again.

Putchar

Function: Puts a character to the screen. Special embedded nonprinting characters are used to control printing functions like boldface, italic, etc.

Accessed: **displayBufferOn**
bit 7 - write to foreground screen if set
bit 6 - write to background screen if set
currentMode - the character is printed in the current style

Pass: a - ASCII character (0-96)
r1H - **y position** for text (0-199)
r11 - **x position** for text (0-319)

Inline Pass: data appears immediately after the jsr

Return: r11 - **x position** for next character
r1H - **y position** for next character

Destroyed: a, x, y, r0, r2 - r10, r12, r13

Synopsis: Putchar prints an ASCII character to the screen at the given x, y position, and may also process any of a number of embedded single byte style/formatting commands. When putting a char to the screen, it updates the position parameters, r11 and r1H. Calling Putchar again with a new character will place the character at the correct position on screen.

There are several multi-byte escape commands supported by PutString which are not supported by Putchar. You need to pass a multibyte string to use these commands and Putchar only accepts a single byte in a. These multi-byte commands, such as GotoXY, are documented in PutString. Font support in GEOS is discussed in detail in the "Font and Text" section of the Commodore 64 Programmer's Reference Guide.

The following special characters are supported:

Char	ASCII	Function
NULL	0	Terminates string
BACKSPACE	8	Erase the previous char whose width is stored in lastWidth. Without support from the application to reload lastWidth, this only works once
FORWARDSPACE	9	Move right the width of space char in current font
LF (line feed)	10	Move down one line (value of currentHeight)
HOME	11	Move to upper left corner of screen
UPLINE	12	Move up one line (value of currentHeight)
CR (carriage rtn)	13	Move to beginning of next line: x position is set to leftMargin, and LF is automatically executed.
UNDERLINEON	14	Turn on underlining
UNDERLINEOFF	15	Turn off underlining
REVERSEON	18	Turn on reverse video
REVERSEOFF	19	Turn off reverse video
BOLDON	24	Turn on boldface
ITALICON	25	Turn on italics
OUTLINEON	26	Turn on outline
PLAINTEXT	27	Return to plain text

If a char is to be drawn outside the position range **leftMargin** - **rightMargin**, then the routine whose address is stored in **StringFaultVector** is called. The application must supply its own **StringFaultVector** routine. If none is provided, then the character is not printed. The user **StringFault** routine often checks **windowBottom** and advances the next line if everything is ok, but can also support word wrap and/or other features. If any part of a character appears outside of **windowTop** and **windowBottom** the character will be clipped so that only the bottom or top part of the character will be drawn.

SmallPutChar is a direct interface to the internal GEOS routines **Putchar** uses. It only updates the x position; it doesn't update boundary checks, or process embedded command characters. You must pass it a printable character. It is useful for applications which must do this checking themselves and want to avoid the time penalties of the redundant checks.

GetRealSize

Function: Returns the size of a character in the current mode (bold, italic...) and current Font.

Pass:

- a - the ASCII character (\$20 - \$60)
- x - the **mode** byte as stored in currentMode

Return:

- y - character **width**
- x - character **height**
- a - **baseline offset**

Destroyed: nothing

Synopsis: GetRealSize returns the height and width of a character when style information is taken into account. The baseline offset is the distance from the baseline to the top of the tallest char in the charset. BaselineOffset = height - descender.

Note: these styles affect the character's size:

Bold: makes character 1 pixel wider.

Outline: makes character 2 pixels higher and therefore also adds two to the baseline offset.

Italics: leans the character over. The effect is of taking a rectangle and leaning it into a parallelogram. The width is not actually changed. The same number of plain text characters will fit on a line as italic characters. Beginning at the baseline, each pair of scanline in the character is shifted over 1 pixel progressively. Thus the baseline is unchanged, the two lines above it are shifted to the right one pixel, the next two are shifted two pixels from their original position and so forth. The lines below the baseline are shifted left.

GetCharWidth

Function: Determine the width of the indicated character in the currentFont and currentMode.

Accessed: currentMode - the current style

Pass: a - an ASCII char (\$20 - \$7E)

Return: a - width of char, or 0 if char was a control character (char < \$20)

Destroyed: y

Synopsis: Returns the width of the char taking into account the current style and font.

Style Control

Printing in bold, italic or any of the other styles is simple. Just pass Puchar the control char for the style you want to display. The style change control chars are listed in the table for Puchar. Puchar won't try to print the unprintable control char, it will simply modify the variable `currentMode` to reflect the new style. Each bit in `currentMode` represents a different style. Passing Puchar a style change control character will add that styles bit to `currentMode`. Passing the style change character for plain text, `PLAINTEXT`, will remove any bold, italic, outline, underline or reverse video mode bits from `currentMode`.

Processing font changes is a little more complicated since you need get the font data for the new font from disk. To do this requires some way of managing fonts in your application. A mechanism for selecting the styles and fonts, e.g., a menu or icon as opposed to just printing pre-existing strings is just as important.

Any string printed by `PutString` containing one of the style escapes will print in the requested style. Each style escape except `PLAINTEXT` adds a style option. `PLAINTEXT` removes all options. As an example, the following text string will print the following string:

TextExample:

```
.byte"Plain","BOLDON","bold","OUTLINEON","Bold and Outline","PLAINTEXT,0
```

Plain,bold,Bold and Outline.

TextExample will also leave `currentMode` indicating plain text.

When entering text with `GetString`, all nonprintable characters are filtered out. If your are retrieving characters from `keyData` and `GetNextChar`, all ASCII values are passed through even bytes having the top bit set by the Commodore key. Thus if you typed the correct ascii control character for `BOLDON` (ASCII 24), then in `OurGetString` above you would actually change the style of characters echoed back to the screen after that. Normally you would want to provide some sort of filtering in `OurGetString` so that all special nonprintable characters are caught before they are sent to Puchar. For example, the tab key sends the code for causing a change in x position. If you were writing a word processor, you might want to have the key sequence Commodore-b cause a change to boldface instead of

having to go to a menu to do that, or trapping the character sent by the tab key to call your tab routine.

Adding font changes into your internal buffer storage for a text string is similar to style changes. A font change string is embedded in the text string. While there is a standard font change string format, there are no GEOS routines that accept them since that would require that the GEOS Kernal be capable of reading fonts from disk. Even compacted, fonts take up enormous amounts of space, too much to include in a c64 OS. The application must provide a way of selecting a font, loading the font in from disk, and entering the font escape string which includes a unique font ID, into the application's internal text buffer. We will now cover the details of changing fonts and then build an example that allows fonts to be switched with a menu selection. The same type of menu scheme can be used to change styles as well.

Fonts

In GEOS, the size of a character is given in points and measures the character's height. One point is normally $1/72$ of an inch. In GEOS, one point is one screen pixel, and one screen pixel is $1/80$ of an inch in order to work best with 80 dot per inch printers. Characters are grouped together into character sets. In GEOS, character sets contain 96 characters (or less if some characters are not defined).^{*} For standard English font the characters normally associated with the ASCII codes 32 to 127 are used. Foreign language character sets will lose some of the special characters such as square bracket in order to provide other characters more important to the language. Each character within a set can have a different width (w, for example, is fatter than i) but all have the same height. There is no limit in GEOS to the height of a character set, however, any single character can be no wider than 54 pixels.

A font is a group of different sized character sets of the same style. The font used by the system in GEOS is BSW and has only one size, 9 point. A specific size (character set) of a font is commonly referred to by name and number, such as BSW 9. The system font is a part of the GEOS Kernal and is always resident in memory. All other fonts are stored on disk and must be loaded by applications as needed.

^{*} One exception: the system character set BSW9 contains 97 characters. ASCII char #128 is the Commodore key char. Font ID

Each font has a unique identification number, font number, for short. Font numbers are 10 bits long. Legal font ID's range from 0 to 1027. Each character set has its own unique font ID. The font ID is a combination of the font number and the character set's point size.

Font ID:

Bits 15-6 = font number; Bits 5-0 = point size.

File Structure

Fonts are stored in VLIR files of of GEOS type FONT. (See the chapter on VLIR files.) Each character set of the font is a record of the VLIR file. The nth point size character set of the font is stored in the nth record of the file, e.g., record 12 contains the 12 point character set. If a record in the font file is empty then the corresponding point size does not exist. Information necessary to use the font is stored in the font file's File Header block. These fields would normally contain information not particularly germane to a font file.

Offset into File Header	# Bytes	Description
OFF_GHFONTID = 128 (\$80)	2	Unique Font ID
OFF_GHPOINT_SIZES = 130 (\$82)	32	Font ID for each character set, from smallest to largest. Padded with 0s.
OFF_GHSET_LENGTHS = 97 (\$61)	32	Size in bytes of each character set from smallest to largest. Padded with 0s.

As shown in the table above, the Font Number for the font is stored in bytes 128 and 129 of the File Header. At an offset of 130 is a 32 byte table containing the Font ID for each character set available for the font. Thus if point sizes 9, 12, and 24 were available, the Font ID Table would contain the three word length entries: 9, 12, and 24. The size in bytes of each of these character sets is stored in the the Point Size Table beginning at byte 97. For our example above, the Point Size Table would also have three word length entries, one word containing the size of each character set. Both the Font ID Table and the Point Size Table are padded with zeros out to the complete 32 bytes so that an application will always know where each table begins. Since both of these tables are 32 bytes long, there can be no more than 16 character sets per font.

Using Fonts

A character set is stored as a contiguous block of data. Once this data is in memory, the character set is used by calling `LoadCharSet` with `r0` pointing to the beginning of the character set data. The system character set is used by calling `UseSystemFont`. An application can use fonts in several different ways. If it needs a specific character set often it can simply read the character set into part of its memory space and leave it there. On the other hand, if an application needs several fonts on the disk, such as `geoPaint` or `geoWrite`, the character sets must be loaded into memory as needed. To do this a method of choosing fonts must be provided. As a standard, both `geoWrite` and `geoPaint` present a font menu for the user to choose his fonts from. This is how it is done: Space for the font menu is left in the menu table structure in the application. More specifically, where the text strings to display as menu selections would normally appear in the menu structure, 19 bytes are left for each of 8 filenames and are initially set to 0. The first two of the 19 characters are for `"* "` which is used to mark the font currently in use. The next 16 characters are for the fontname, and the last character is for the null terminator, 0.

When the application's initialization routine is run, `FindFTypes` (See the file system chapter) is called to get the names of files of type `FONT`. The filenames returned by `FindFTypes` are then copied into the menu structure. When the font menu is opened, the fonts available on the disk will appear.

Next the Font ID tables and Point Size Tables are retrieved from the File Header blocks of each font file and stored away so that the point size menu can be constructed and each character set may be checked to make sure it fits into the memory space allocated. The point size menu is built dynamically by using a `DYNAMIC_SUB_MENU`. The font menu structure is set up so that each of the fonts menu items are of type `DYNAMIC_SUB_MENU` and each point to the same dynamic sub-menu routine. When any of the font sub-menus are selected, the same dynamic sub-menu routine is called. This routine then gets the information it needs and builds the point size menu for the appropriate font.

The routine builds the point size menu as follows. First it needs the name of the font currently selected. It takes the number of the selected menu item as passed by the GEOS Kernal in `a` and uses it to compute which font is being selected. It indexes into the menu table to pull out the name string. This will be use to load the font file. Next the point size tables as stored away by the application's initialization routine are used. The number of the menu item (as was passed in `a`) is used to determine which set of point sizes corresponds

to the font selected. The point size table for the font selected is then used to create the point size menu structure.

When the point size is selected your point size routine takes the menu item number (again passed in *a*) and figures out which the point size was selected, and loads the correct record in the font file. For example, when the user pulls down the font menu and selects Roma, the dynamic submenu routine checks *a* and figures out that retrieves the string Roma from the menu text area. It then checks the point sizes for Roma that the initialization code saved, and enters those into the point size menu structure, making sure to enter the correct number into number of items field in the menu table. The dynamic submenu actually goes in and modifies the menu table before the GEOS Kernal draws the menu on screen. Next, after the dynamic submenu routine finishes and the point size menu is displayed, the user selects a point size and the menu action routine pointed to by all the point size menu items is called. The menu action routine receives the number of the menu selected in *a* and uses it to construct an index into the point size menu table to figure out what point size the user is selecting.

We now have the filename for the font and the record number to load. We call `OpenRecordFile`, with the filename, namely Roma, and the point size is passed as the record number. Once the character set is in memory, `LoadCharSet` may be called to load the font.

A list of all the currently available fonts appears in the Appendix.

LoadCharSet

Function: Changes the character set being used to draw characters.

Pass: r0 - address of FontInfoTab to use

Return: nothing

Destroyed: a, y, r0

Synopsis: Fonts are loaded from disk and require between 733 and 3970 bytes in memory. Several fonts may be loaded into memory at the same time. LoadCharSet is called to tell GEOS to activate a font. If the font to be activated was loaded at \$3000 then this is the value to pass in r0.

UseSystemFont

Function: Return character drawing to BSW9 font

Pass: nothing

Return: nothing

Destroyed: a, y, r0

Synopsis: The system font, BSW-9 is always in memory. Since there is presently no way to tell what font is currently loaded, UseSystemFont is provided as a quick way to get back to the standard font.



7.

Input Driver

The Standard Driver

GEOS currently supports the joystick (the standard driver), a proportional mouse, and a graphics tablet. On the screen, the position of the joystick or mouse is shown by an arrow cursor. We shall use the terms, mouse, pointer, and cursor, interchangeably to refer to the arrow cursor on the screen. We shall use the term device to denote the actual hardware.

Each Interrupt, the GEOS Kernal Interrupt Level code calls the input driver. The job of the input driver is to compute the values of the following variables.

<code>mouseXPosition</code>	Word	X position in visible screen pixels of the mouse pointer (0-319)
<code>mouseYPosition</code>	Byte	Y position in visible screen pixels of mouse pointer (0-199)

<code>mouseData</code>	Byte	Set to nonnegative if fire-button pressed, negative if released.
<code>pressFlag</code>	Byte	Bit 5 (<code>MOUSE_BIT</code>) set if a change in the button, Bit 6 (<code>INPUT_BIT</code>) if any change in input device since last interrupt.

Both the GEOS Kernal and applications may then read and act on these variables. The GEOS Kernal reads bit 5 (`MOUSE_BIT`) in the `pressFlag` variable to determine if there has been a change in the mouse button. If there has been a change, then the Kernal reads `mouseData` to determine whether the change is a press or release. If the mouse button has been pressed (indicated by `mouseData` changing from negative to nonnegative) then GEOS will check to see whether the mouse position is over a menu, an icon, or screen area. If it is over a menu, then the menu dispatcher is called. If its over an icon, then the icon dispatcher is called. If its not a menu or icon then the routine in `otherPressVector` is called.

If the joystick changes from being pressed to being released (`mouseData` has a negative value) then the Kernal will vector through `otherPressVector`. Note: all releases are vectored through `otherPressVector`, even if the original press was over a menu or icon. The application's `otherPressVector` routine must be capable of screening out these unwanted releases. The reason that the mouse acts like this is that the ability to detect releases was added relatively late to the GEOS Kernal. The menu and icon modules were already complete. `otherPressVector` is called on all releases including those for menus and icons so that its routine can take special action on those releases as well as its own, if necessary. Usually, the application's `otherPressVector` routine will either ignore releases altogether, or only act on releases following screen area presses.

What an Input Driver Does

It is the job of the input driver to read the hardware bytes it needs to load `mouseData` and `pressFlag` with the proper values. It must determine the change in the position of the mouse and store new values in `mouseXPosition` and `mouseYPosition`.

Different input drivers compute the mouse x, y position in entirely different ways. As an example, the joystick driver does this by first reading the joystick port, and then com-

puting an acceleration from the direction the joystick was pressed. From that, a velocity, and finally a position are determined. A proportional mouse is entirely different. The Commodore mouse sends differing voltage levels to the potentiometer inputs in the joystick port and the SID chip in the c64 reads the voltage level and stores an 8 bit number for both x and y. The driver computes a change in position from the voltage level as reflected by the value of the two bytes. No matter how it is done, though, the input driver is responsible for setting the 4 variables mentioned above.

Location of Responsibilities of Input Driver

The code for the joystick input driver takes up the 380 bytes beginning at `MOUSE_BASE`, the area from `$FE80-$FFF9`. When an alternate input driver such as a graphics tablet is loaded by the deskTop, it is intalled at this location. If you write an input driver, it should be assembled at this address. All GEOS applications will expect three routines, `InitMouse`, `SlowMouse` and `UpdateMouse`, and the four variables mentioned above to be supported by any input driver. These three routines should perform the same function, regardless of the input device. This way the particular application running need know nothing about which input driver the user has chosen. These routines may begin anywhere within the input driver area just so long as a short jump table is provided right at the beginning of the input driver space:

<u>Address</u>	<u>Contents</u>
<code>MOUSE_BASE</code>	<code>jmp InitMouse</code>
<code>MOUSE_BASE + 3</code>	<code>jmp SlowMouse</code>
<code>MOUSE_BASE + 6</code>	<code>jmp UpdateMouse</code>

These are the addresses that the GEOS Kernal and applications will actually call. For example, to call `UpdateMouse`, the Kernal will do a `jsr MOUSE_BASE + 6` during Interrupt Level. The first routine the input driver must provide is `InitMouse`. It is called to perform any initialization, and set any variables, the driver needs before the other two routines are called.

InitMouse

Function: Perform hardware dependent initialization.

Pass: nothing

Return: mouseXPosition - set to initial values (0-319)
mouseYPosition - set to initial values (0-199)
mouseData - set to released (negative value)

Destroyed: a, x, y, r0 - r15 - assume all registers destroyed unless the input driver being used is known.
a, x, y, r0 - r2, r7 - r8 - for joystick

Synopsis: Does whatever initialization is necessary for the input device. This includes setting any internal variables needed by the driver to initial values.

After the call to InitMouse, it should be possible to execute SlowMouse and Update Mouse.

Acceleration, Velocity, and Nonstandard Variables

Some input devices, such as the joystick, need to be adjusted for different sensitivities. For example, sometimes the user will want the joystick to accelerate to its maximum velocity quickly. Other times, such as when opening a menu, the user will want it to move more slowly so as to make it easier to select an item without slipping off the menu altogether.

Other devices such as proportional mice and graphics tablets do not make use of acceleration and velocity. These devices deal more directly with position and distance moved. Still other devices as yet uninvented may need special variables of their own. The question arises how to best support different input devices in a way that the application need not know which device is being used, and yet leave room for new devices. There are three parts to the solution.

First, there is a basic level that every input drive should be able to support. This includes maintaining the position variables `mouseXPosition`, and `mouseYPosition`, and the mouse button variables, `pressFlag`, and `mouseData`. At the very least, an input driver must generate values for these variables.

Second, additional variables for joystick-like devices, are allocated in the GEOS Kernal RAM space. The joystick is the default driver for GEOS, and needs to keep track of acceleration and velocity variables. These variables include `maximumMouseSpeed`, `minimumMouseSpeed`, and `mouseAcceleration`. These variables are loaded with default values by the driver's initialization routine, and are located in GEOS Kernal RAM area so that they may be used by the preference manager to adjust the speed of the mouse. There is also a routine, `SlowMouse` that is called by the GEOS Kernal itself to slow the mouse down during menu selection. This routine is presented below. Together this routine and these variables allow a high level of control over joystick behavior. This may seem like a lot of effort to spend on a joystick, but considering that most users will be using a joystick, such effort is appropriate.

Different devices like Commodore's proportional mouse do not require any special treatment. It is not based on velocity, but on distance. Its motion is precise enough to make fine tuning unnecessary. It is possible that some as yet unknow input device may become available that does requires special treatement. In this case a third approach may be used.

This approach is to augment the regular position and button variables with four bytes beginning at the label `inputData` in the Kernal RAM. These variables may be used to pass additional values to an application. Any input device that needs to pass parameters to an application other than the position, mouse button, or velocity and acceleration variables, should pass them here. Note: Applications which rely on `inputData` become device dependent.

Whenever the input state has changed, the driver must:

1. update the 4 mandatory mouse variables;
2. update `inputData`, if supported;
3. the `INPUT_BIT`, (bit 6) should be set in `pressFlag`.

In addition, an application that uses `inputData` must load the vector `inputVector` with the address of a routine that retrieve values from `inputData`. When the Kernal sees the `INPUT_BIT` set, it will vector through `inputVector` if it is nonzero. As an example, the joystick driver loads a value for the direction in the first of these four bytes and the current speed of the mouse in the second. `geoPaint` uses these values in its routine to scroll the drawing. When in `scrollMode`, `geoPaint` sets `inputVector` with the address of a routine used in scrolling. Whenever the direction of the joystick changes, `inputVector` is vectored through and the `geoPaint` scroll routine stops or changes the direction of the scrolling.

This use of these variables is probably unfortunate because although they are natural to generate for the joystick, they are not so natural to generate for other drivers, such as proportional mice. The drivers for these devices must generate these direction values by hand so that they will completely work with `geoPaint`.

The only reason for using `inputData` is to support a special input device that communicates in a custom fashion with its own application. As this can cause incompatibility with other input devices and other applications, this approach should be used sparingly.

An application can check the variable string `inputDevName` for the name of the current input device. The `deskTop` loads the null-terminated filename of the input driver file into this 17 byte string.

The general approach then for supporting a new input driver should be clear. First compute the position and button variables. If geoPaint scrolling is to be supported, direction variables will need to be supported. Finally, some custom tailorable driver support is possible. The variables discussed above are presented in more detail below, after the outlines for Slow Mouse and UpdateMouse.

SlowMouse

The SlowMouse routine, as outlined below, sets the joystick speed to zero. The joystick is then free to accelerate again. From its name, one might instead expect SlowMouse to reduce the maximumMouseSpeed, but this is not the case.

The reason for having a routine like this is to make using menus easier. When a menu opens, and the user slides down the selections and hits the mouse button when over the desired item. The GEOS Kernal will then open a submenu and put the mouse pointer on the first selection of the submenu. The user may then select one of it's items. It was found that almost all users keep the joystick direction pushed until the submenu comes up. By this time the mouse will have reached maximum velocity, and, when placed on the submenu graphic by the application, will go flying off. SlowMouse just zeros out the mouse's speed so that this won't happen. Drivers for mice and graphics tablets which don't use velocity need to include this routine even though in this case it will merely perform an rts.

To make the mouse actually slow down from within an application, maximumMouseSpeed, and mouseAcceleration can be lowered. The standard values for these variables may be found in the Mouse Variable and Mouse Constant sections later in this section.

SlowMouse

Function: Make the input device more sensitive, i.e. , slow it down.

Pass: nothing

Return: nothing

Destroyed: a - for joystick driver
nothing - for most other devices, usually best to assume
a, x, y, r0 - r15 destroyed

Synopsis: When moving onto a menu it is sometimes necessary to **stop** the mouse cursor. This routine doesn't reduce the maximumMouseSpeed as one might expect, but instead, for the standard joystick driver, it sets the mouse speed to zero. The mouse is then free to accelerate again.

The versions of SlowMouse provided by Graphics tablets and true proportional mice don't actually do anything but rts.



UpdateMouse

UpdateMouse is the main routine in an input driver. Its responsibilities include reading the joystick port in order to determine how the input device has changed, and translating this into a change in `mouseXPosition`, `mouseYPosition`, `mouseData` and `pressFlag`. If `geoPaint` scrolling is to be supported, then direction information must be returned in `inputData`. If special input driver information is to be passed to an application then `inputData` should again be used.

UpdateMouse

Function: Update the mouse controlled variables such as position and fire button. Called every interrupt by GEOS Kernal Interrupt Level.

Pass: mouseXPosition - current **XPosition** of input device
 mouseYPosition - current **YPosition** of input device
 cia1prb - port for joystick

Return: mouseXPosition - new **XPosition** of input device
 mouseYPosition - new **YPosition** of input device
 mouseData - nonnegative for press, negative for release
 pressFlag - MOUSE_BIT set if fire button changed state
 INPUT_BIT set if status stored in inputData changed state

inputData - four optional extra data bytes for device dependent status.
 For joysticks, stores the joystick direction.

inputData: 0 -7
 joystick directions:
 0 = right
 1 = up & right
 2 = up
 3 = up & left
 4 = left
 5 = left&down
 6 = down
 7 = down&right
 - 1 = joystick centered

inputData+1: current mouseSpeed

Destroyed: a, x, y, r0 - r15 for joystick

Synopsis: This routine is called every interrupt to update the position of the input device and the status of the fire button. mouseXPosition and mouseYPosition are updated each interrupt. When the fire button changes, the MOUSE_BIT bit in pressFlag is set. mouseData becomes nonnegative if the fire button is pressed, and negative if released. If the MOUSE_BIT has been set by UpdateMouse, then GEOS will read mouseData during Main Loop to determine if the change was a press or release. GEOS will then vector to the proper dispatch routine -- icon, menu, or otherPress-

Vector -- depending on the location of the mouse and state of the button when the button was pressed. Upon return from this dispatch routine, the `MOUSE_BIT` will be reset to 0.

Any special data bytes supported by the input device should be stored in the `inputData` array. The `INPUT_BIT` in `pressFlag` should be set to when there is a change in `inputData`. During `MainLoop`, GEOS will vector through `inputVector`, if the `INPUT_BIT` is set and `inputVector` is nonzero.

Mouse Variables for Input Driver

The following variables are supported by the mouse module. Most of these variables have been described briefly above.

Required Mouse Variables

<code>mouseXPosition</code>	Word	X position in visible screen pixels of the mouse pointer (0-319)
<code>mouseYPosition</code>	Byte	Y position in visible screen pixels of mouse pointer (0-199)
<code>mouseData</code>	Byte	Nonnegative if fire-button pressed, negative if released.
<code>pressFlag</code>	Byte	Bit 5 (<code>MOUSE_BIT</code>) set by driver if a change in the button; Bit 6 (<code>INPUT_BIT</code>) set if any change in input device since last interrupt.

```

MOUSE_BIT    =    %00100000
INPUT_BIT    =    %01000000

```

Optional Mouse Variables

<code>maximumMouseSpeed</code>	Byte	Used to control the maximum speed or motion of the input device. In the case of a joystick, <code>maximumMouseSpeed</code> controls the maximum velocity the mouse can travel across the screen. This variable is unused for graphics tablets and proportional mice. Best values for this byte depend on how the input driver uses this variable to compute current speed and position. For a joystick legal values are 0-127. Default value is:
--------------------------------	------	--

```
MAXIMUM_VELOCITY=127
```

This is the constant for the default `maximum velocity` to store in `maximumMouseSpeed`

<code>minimumMouseSpeed</code>	Byte	Used to control the minimum speed or motion of the input device. See <code>maximumMouseSpeed</code> above. Legal joystick values are: 0-127. Default value is:
--------------------------------	------	--

```
MINIMUM_VELOCITY=30
```

Minimum velocity to store in `minimumMouseSpeed`. Anything slower than this bogs down.

mouseAcceleration Byte

This byte controls how fast the input device accelerates. In the case of a joystick, it controls how fast the joystick accelerates to its maximum speed. In the case of a graphics pad it might scale the distance moved with the pointer on the pad to the distance moved on the screen. Currently this variable is only used by the joystick driver. Legal values are 0-255. Default value is:

MOUSE_ACCELERATION=127

Typical acceleration byte value of mouse

inputVector

Word

Contains the address of a routine called from MainLoop to use input driver information supplied by unorthodox input devices. The idea here is that some input drivers may be able to produce more information than the x and y position data for an application may want to use this info. If UpdateMouse supports such extra info it should store it in inputData array and set the INPUT_BIT in pressFlag. When GEOS MainLoop sees this bit set it will call the routine whose address is stored in inputVector.

inputData

4 Bytes

Used to store device dependent information. For joysticks,
inputData:0-7

joystick directions:

- 0 = right
- 1 = up & right
- 2 = up
- 3 = up & left
- 4 = left
- 5 = left&down
- 6 = down
- 7 = down & right
- 1 = joystick centered

inputData+1: current mouseSpeed

The Mouse as Seen by the Application

To this point, we have discussed input devices as seen from the perspective of a programmer wanting to write an input driver. The other side of the coin is how an application interacts with the input driver. The regular action of the mouse is as described above. Mouse presses are checked for icon, or menu activation, or a press in the user area of the screen.

To start the mouse functioning like this, the routine `StartMouseMode` is called. Since this is done by the `deskTop` to get itself running, the application need not call `StartMouseMode` itself. To turn mouse functioning off, one calls `ClearMouseMode`. A bit in the variable `mouseOn` is cleared, the sprite for the mouse is disabled (the sprite data is no longer DMA'd for display, important for RS-232, disk, and other time critical applications) and `UpdateMouse` is no longer called during interrupt level. This is the reason the mouse pointer flickers during disk accesses: `ClearMouseMode` is called by the disk turbo code. To restore mouse functioning after a call to `ClearMouseMode`, call `StartMouseMode`.

To temporarily turn the mouse picture off, but have its position and `inputData` variables still set, call `MouseOff`. `UpdateMouse` in the input driver is still called, just the sprite for the mouse, sprite 0 is disabled. To turn the mouse on again, call `MouseUp`. `MouseUp` reenables the mouse sprite and causes the mouse to be redrawn the next interrupt in case the mouse had been moved since being turned off. To temporarily disable the mouse, call `MouseOff` and then `MouseUp`.

StartMouseMode

Function: Turn the mouse functioning on. Calls InitMouse.

Pass: carry - set to store initial positions and slow velocities down
r11 - x position (0-319)
y - y position (0-199)

Return: nothing

Destroyed: a, x, y, r0 - r15

Synopsis: StartMouseMode turns the functioning of the mouse on. mouseVector is activated. The global variables mouseLeft, mouseRight, mouseTop and mouseBottom must be set already. StartMouseMode is usually called by the application's init code after any changes if any to the mouse speed variables have been made.

If carry is set then the positions passed in y and r11 are loaded into mouseXPosition and mouseYPosition and SlowMouse is called to set small velocities for the joystick.

Another routine, InitMouse, is part of the input driver module and is automatically called at GEOS boot time, and deskTop boot time, to initialize positions, velocities and any other variables needed by the input device.

MouseOff

Function: Temporarily disable mouse.

Pass: nothing

Return: nothing

Destroyed: a, x, y, r3L

Synopsis: Turns the mouse off and erases the mouse cursor. All of the mouse variables retain their values. The mouse can be temporarily disabled by calling mouseOff and then mouseUp.

MouseUp

Function: Force the mouse cursor to be drawn, reactivate mouse after calling mouseOff.

Pass: nothing

Return: nothing

Destroyed: a

Synopsis: Turns the mouse on after a call to mouseOff and sets a flag telling GEOS to redraw the mouse cursor at the next interrupt. The mouse can be temporarily disabled by calling mouseOff and then mouseUp.

Additional Mouse Control

GEOS allows you to limit the movement of the mouse to a region on screen. The GEOS Kernal will constrain the mouse withing a rectangle defined by two word length variables, `mouseLeft`, and `mouseRight`, and two byte length variables, `mouseTop`, and `mouseBottom`. The input driver need know nothing about these variables. After it updates `mouseXPosition`, and `mouseYPosition`, the Kernal will check to see if the new position is out of bounds, and if necessary force its position back to the edge of the rectangle. The Kernal will also vector through `mouseFaultVector`. This vector is initialized to zero by the Kernal. The application may load `mouseFaultVector` with the address of a routine to implement, for example, scrolling a document under the screen window. The effect would of the screen scrolling whenever the user drew the mouse pointer off the edge of the screen.

There is also a routine for checking to see if the mouse pointer is within a certain region on screen. This routine is quit useful if clicking inside a box or other region is to have special significance in your application. This routine is called `IsMseInRegion` and you pass it the coordinates of the sides of the rectangular region you want it to check.

A couple of more mouse variables are used. `mousePicData` contains 64 bytes for the sprite picture of the mosue, while `mouseVector` contains the address of the routine `MainLoop` calls to handle all mouse functioning. If the `MOUSEON_BIT` of `mouseOn` is set, then every time the input driver indicates the mouse button has been pushed, `mouseVector` is vectored through. It is unclear why the programmer might want to change `mouseVector`, as this would disable icon and menu handling. `otherPressVector` is more likely the vector to change.

`mouseOn` also contains bits for turning menu and icon handling on and off. Unfortunately, a call to the menu handling routine will serve to turn the icon enable bit on upon its exit. This is the reason a dummy icon table is necessary for those running without icons.

These variables and the constants used to set them are described below immediately following `IsMsInRegion`.

IsMseInRegion

Function: Tests if a mouse is inside the indicated region.

Pass: r2L - y coordinate of **top** of region (0-199)
r2H - y coordinate of **bottom** of region (0-199)
r3 - x coordinate of **left** edge of region (0-319)
r4 - x coordinate of **right** edge of region (0-319)

Return: a - TRUE (-1) if in region, FALSE (0) if outside

Destroyed: nothing

Synopsis: IsMseInRegion tests to see if the x, y position of the mouse is within the region defined by the rectangle passed in r2 - r4. If it is, then TRUE is returned in a else FALSE is returned.

Mouse Variables for Applications

The following variables are supported by the mouse module in the GEOS Kernal for application use.

mouseOn

Byte A flag which contains bits determining the status of the mouse, and menus.

Also contains bits used by the Menu and Icon modes.

bit 7 Mouse On if set
 bit 6 Set if Menus being used (should always be 1)
 bit 5 Set if Icons being used (should always be 1)

SET_MOUSEON = %10000000 Bit set in mouseData to turn mouse on
 SET_MENUON = %01000000 Bit set in mouseData to turn Menus on
 SET_ICONSON = %00100000 Bit set in mouseData to turn Icons on
 MOUSEON_BIT = 7 The number of bit used to turn mouse on
 MENUON_BIT = 6 The number of bit used to turn on menus
 ICONSON_BIT = 5 The number of bit used to turn on icons
 A bug causes menu functioning to turn this bit back on.

mouseLeft

Word mouse cursor not allowed to travel left of this programmer set position. Legal range is 0-319.

mouseRight

Word Mouse cursor not allowed to travel right of this pixel position on screen. Legal range is 0-319.

mouseTop

Byte Mouse cursor not allowed to travel above this pixel position on screen. Legal range is 0-199.

mouseBottom

Byte Mouse cursor not allowed to travel below this this pixel position on screen. Legal range is 0-199.

mousePicData

64 Bytes Sprite picture data for mouse cursor picture. This area is copied into the actual sprite data area by the GEOS Kernal.

mouseVector

Word Routine called by GEOS Kernal when mouse button pressed.

mouseFaultVector

Word Routine to call when mouse tries to go outside of mouseTop, Bottom, Left, and Right boundaries. GEOS will not allow the mouse to actually go outside the boundaries.

Joystick

This file contains the joystick input driver.

```
.if(0)
```

Callable Routines:

```
o_InitMouse  
o_SlowMouse  
o_UpdateMouse
```

```
.endif
```

```
.include Macros6500  
.include Macros_test  
.include Constants  
.include Memory_map  
.include Routines
```

```
.psect $400
```

```
;the file header is assembled at $400 for those using  
;PRGTOGEOS to createa GEOS file from a c64 SEQ
```

JoyName:

JoyHdr:

```
.byte %11111111,%11111111,%11111111
.byte %10000000,%00000000,%00000001
.byte %11000000,%11001111,%11110001
.byte %10100011,%00110000,%00001001
.byte %10011100,%00000001,%10000101
.byte %10000000,%01100010,%01000101
.byte %10000000,%10011100,%00100101
.byte %10000001,%01101100,%00100101
.byte %10000010,%11100100,%00011001
.byte %10000100,%11011010,%00000001
.byte %10001000,%00111010,%00000001
.byte %10010000,%00110110,%00000001
.byte %10100000,%00001100,%11111001
.byte %10100000,%00011000,%10000101
.byte %10110000,%00110000,%10000101
.byte %10011000,%01100000,%11111001
.byte %10001100,%11000000,%10000001
.byte %10000111,%10000000,%10000001
.byte %10000011,%00000000,%10000001
.byte %10000000,%00000000,%00000001
.byte %11111111,%11111111,%11111111
```

```
.byte $80|USER ;Commodore file type assigned to GEOS files
.byte INPUT_DEVICE ;GEOS file type
.byte SEQUENTIAL ;SEQ file structure
.word MOUSE_BASE ;start address for saving file data
.word endJoystick ;end address for saving file data (-1)
.word 0 ;not actually used (execution start address)
.byte "Input Drvr V1.1",0,0,0,0 ;20 byte permanent name
;20 bytes for author name
.byte "Dave & Mike ",0,0
;20 bytes for parent appl. name (not used)
.byte 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

```
.if (0)
```

Jump Table to MouseDriver Routines

```
.endif
```

```
.psect MOUSE_BASE
.psect
```

```
;
; Input driver jump table
;
```

```
    jmp    o_InitMouse      ;entry #0
    jmp    o_SlowMouse      ;entry #1
    jmp    o_UpdateMouse    ;entry #2
```

```
;
; Global variables:
```

```
diskData      =      inputData      ;current disk direction
mouseSpeed    =      inputData + 1   ;current mouse speed
```

```
;
; Local variables:
```

```
fracXMouse:      ;fractional mouse position
    .byte    0
fracYMouse:      ;fractional mouse position
    .byte    0
fracSpeedMouse:  ;fractional part of current mouse speed
    .byte    0
velXMouse:       ;x component of currentSpeed
    .byte    0
velYMouse:       ;y component of currentSpeed
    .byte    0
currentMouse:    ;current value of fire button
    .byte    0
currentDisk:     ;current value of joystick
    .byte    0
lastKeyRead:     ;for debouncing joystick
    .byte    0
```

```
.if(0)
```

InitMouse

Synopsis: Internal routine: This routine initializes the 'mouse'.

Called By: At initialization EXTERNALLY

Pass: mouseXPosition, mouseYPosition - starting position for the mouse

Return: none

Accessed: none

Destroyed: a, x, y, r0 - r15

```
.endif
```

```
o_InitMouse:
    jsr    o_SlowMouse           ; do    lda #0
                                ;      sta mouseSpeed
    sta    fracSpeedMouse
    sta    mouseXPosition
    sta    mouseXPosition+1
    sta    mouseYPosition
    lda    #-1                   ;pass release
    sta    diskData
    jmp    ComputeMouseVels      ;store the correct speeds
```

.if(0)

SlowMouse

Synopsis: Internal routine: Called when menus are pulled down to slow the mouse

Called By: InternalOnly

Pass: none

Return: a - 0

Accessed: none

Destroyed: a, x, y, r1 - r13

.endif

o_SlowMouse:
 LoadB mouseSpeed,0 ;zero speed
SM_rts:
 rts

```
.if (0)
```

`.if(0)`

UpdateMouseY

Synopsis: Internal routine: Update the y position of the mouse by adding in the velocity.

Called By: o_UpdateMouse

Pass: mouseYPosition - current y position of the mouse

Return: mouseYPosition - updated

Accessed: none

Destroyed: a, x, y, r1H

`.endif`

UpdateMouseY:

	<code>ldy #0</code>	<code>;assume positive velocity</code>
	<code>lda velyMouse</code>	<code>;get velocity</code>
	<code>bpl 10\$</code>	
	<code>dey</code>	<code>;if negative then sign extend with -1</code>
10\$:	<code>sty r1H</code>	<code>;store high byte</code>
	<code>asl a</code>	<code>;shift left thrice</code>
	<code>rol r1H</code>	
	<code>asl a</code>	
	<code>rol r1H</code>	
	<code>asl a</code>	
	<code>rol r1H</code>	
	<code>add fracYMouse</code>	<code>;add fractional position</code>
	<code>sta fracYMouse</code>	<code>;store new fractional position</code>
	<code>lda r1H</code>	<code>;get high byte of velocity</code>
	<code>adc mouseYPosition</code>	<code>;add position</code>
	<code>sta mouseYPosition</code>	
20\$:	<code>rts</code>	

```
.if(0)
```

UpdateMouseVels

Synopsis: Internal routine: Update the velocity of the mouse by adding in the acceleration

Called By: o_UpdateMouse

Pass: mouseSpeed - current mouse speed
velXMouse, velYMouse - current velocity

Return: mouseSpeed, velXMouse, velYMouse - updated

Accessed: none

Destroyed: a, x, y, r0 - r2

```
.endif
```

UpdateMouseVels:

```

    ldx    diskData           ;get direction
    bmi    20$               ;if release then branch
    lda    maximumMouseSpeed ;check for maximum speed
    cmp    mouseSpeed
    blt    15$               ;if max then do nothing
    lda    mouseAcceleration ;add acceleration to speed
    add    fracSpeedMouse
    sta    fracSpeedMouse
    bcc    30$
    inc    mouseSpeed        ;increment mouse speed if necessary
    bra    30$

15$:
    sta    mouseSpeed

20$:
    lda    minimumMouseSpeed ;get speed
    cmp    mouseSpeed        ;don't make less than minimum
    bge    25$               ;if < minimum then branch
    lda    fracSpeedMouse    ;subtract acceleration from speed
    sub    mouseAcceleration
    sta    fracSpeedMouse
    bcs    30$
    dec    mouseSpeed        ;decrement mouse speed if necessary
    bra    30$

```


25\$:

sta mouseSpeed

30\$:

; jmp ComputeMouseVels

; and fall through to ComputeMouseVels
; Finally, based on direction and
; Speed, calculate new mouse X & Y
; Velocities.

.if(0)

ComputeMouseVels

Synopsis: Internal routine: Compute mouse velocity based on joystick direction

Called By: InternalOnly

Pass: diskData - joystick direction
mouseSpeed - current mouse speed

Return: velXMouse, velYMouse - set depending of passed direction

Accessed: none

Destroyed: a, x, y, r0 - r2

.endif

ComputeMouseVels:

```
    ldx    diskData
    bmi    10$                ;if release then handle
    MoveBy mouseSpeed,r0L     ;pass magnitude
    jsr    SineCosine
    MoveB  r1H,velXMouse
    MoveB  r2H,velYMouse
    rts
```

10\$:

```
    lda    #0                ; if release,
    sta    velXMouse          ; zero x velocity
    sta    velYMouse          ; zero y velocity
    rts
```

```
.if(0)
```

UpdateMouseX

Synopsis: Internal routine: Update the x position of the mouse by adding in the velocity

Called By: o_UpdateMouse

Pass: mouseXPosition - current x position of the mouse

Return: mouseXPosition - updated

Accessed: none

Destroyed: a, x, y, r11, r12L

```
.endif
```

```
UpdateMouseX:
    ldy    #$ff                ;assume negative
    lda    velXMouse
    bmi    10$                ;if indeed negative then branch
    iny
10$:
    sty    r11H
    sty    r12L
    asl    a                    ;multiply by 8 for permanent speed power of 3
    rol    r11H
    asl    a
    rol    r11H
    asl    a
    rol    r11H
                                ;add velocity to fractional position
    add    fracXMouse          ;add fractional position
    sta    fracXMouse          ;store new fractional position
    lda    r11H                ;get high byte of velocity
    adc    mouseXPosition      ;add low byte of position
    sta    mouseXPosition      ;and store
    lda    r12L                ;this is actually triple precision math
    adc    mouseXPosition+1    ;add the high byte of integer x position
    sta    mouseXPosition+1    ;r11 now has newly calculated x position
    rts
```

```
.if(0)
```

C64Joystick

Synopsis: Internal routine: Read the joystick and update the appropriate mouse related variables.

Called By: o_UpdateMouse

Pass: none

Return: none

Affects:

- lastKeyRead - set to new joystick read
- currentDisk - set to new joystick direction (only if new)
- pressFlag - MOUSE_BIT set if fire button pressed
- DISK_BIT set if joystick direction changed
- diskData - new disk direction, if changed
- mouseData - new state of fire button, if changed

Destroyed: a, x, y

```
.endif
```

C64Joystick:

```
LoadB    cialpra,%11111111    ;scan no rows, so we're sure of stick
lda      cialprb              ;get port data for joystick A (port 1)
eor      #$ff                 ;complement data for positive logic
cmp      lastKeyRead          ;software debounce, must be same twice
sta      lastKeyRead          ;store value for debounce.
bne      20$                  ;if not same, don't pass return value

and      #$f                  ;isolate stick bits.
cmp      currentDisk          ;compare to current stick value
beq      10$                  ;if no change then branch...
sta      currentDisk          ;set to new stick value
tay      ..                   ;put value in y
lda      directionTable,y     ;get the value to pass from table
sta      diskData
smbf     INPUT_BIT,pressFlag  ;mark that input device has changed
jsr      ComputeMouseVels
```

```
10$:
```

```
lda      lastKeyRead          ;get press
and      #%00010000          ;isolate the fire button
```

```
    cmp     currentMouse      ;and compare it to the current value
    beq     20$               ;if no change then branch...
    sta     currentMouse      ;else, set new button value
    asl     a                 ;shift into bit 7
    asl     a
    asl     a
    eor     %#10000000        ;complement to pos logic
    sta     mouseData
    smbf    MOUSE_BIT,pressFlag ;set changed bit

20$:
    rts
```

directionTable:

```
.byte -1      ;pass a -1 if no direction pressed.
.byte 2       ;see hardware description at start
.byte 6       ;of this module to understand the
.byte $FF     ;direction conversions here.
.byte 4       ;note that $FF's are nonvalid states,
.byte 3       ;actually they should be impossible
.byte 5       ;unless the controller is broken.
.byte $FF
.byte 0
.byte 1
.byte 7
.byte $FF
.byte $FF
.byte $FF
.byte $FF
.byte $FF
```

.if(0)

SineCosine

Synopsis: Internal routine: SineCosine does a sixteen direction sine and cosine and multiplies this value by a magnitude

Called By: ComputMouseVels

Pass: x, diskData - direction (0 to 15)
r0L - magnitude of speed

Return: r1H - x velocity
r2H - y velocity

Accessed: none

Destroyed: a, x, y, r0, r6 - r8 destroyed

.endif

SineCosine:

```

        lda    cosineTable,x          ;save cosine value
        sta    r1L
        lda    sineTable,x            ;save sine value
        sta    r2L
        lda    sineCosineTable,x      ;get signs
        pha
        ldx    #r1L                   ;compute x velocity
        ldy    #r0L                   ;must do MultBB manually
        jsr    BBMult                  ;to avoid call to BBMult
        ldx    #r2L                   ;compute y velocity
        jsr    BBMult                  ;y already points to r0L
        pla                                         ;to avoid call to BBMult
        pha
        bpl    10$                    ;if x positive then branch
        NegateW r1
10$:
        pla
        and    #%01000000
        beq    20$                    ;if y positive then branch
        NegateW r2
20$:
        rts

```

cosineTable:

```
.byte 255 ;dir 0 - 0 degree angle
.byte 181 ;dir 2 - 45 degree angle
;Note: the cosineTable overlaps the sine table
```

sineTable:

```
.byte 0 ;dir 0 - 0 degree angle
.byte 181 ;dir 2 - 45 degree angle
.byte 255 ;dir 4 - 90 degree angle
.byte 181 ;dir 6 - 135 degree angle
.byte 0 ;dir 8 - 180 degree angle
.byte 181 ;dir 10 - -135 degree angle
.byte 255 ;dir 12 - -90 degree angle
.byte 181 ;dir 14 - -45 degree angle
```

sineCosineTable:

```
.byte POSITIVE | (POSITIVE >> 1) ;dir 0 - 0 degree angle
.byte POSITIVE | (NEGATIVE >> 1) ;dir 2 - 45 degree angle
.byte POSITIVE | (NEGATIVE >> 1) ;dir 4 - 90 degree angle
.byte NEGATIVE | (NEGATIVE >> 1) ;dir 6 - 135 degree angle
.byte NEGATIVE | (POSITIVE >> 1) ;dir 8 - 180 degree angle
.byte NEGATIVE | (POSITIVE >> 1) ;dir 10 - -135 degree angle
.byte POSITIVE | (POSITIVE >> 1) ;dir 12 - -90 degree angle
.byte POSITIVE | (POSITIVE >> 1) ;dir 14 - -45 degree angle
```

Sprite Support

The GEOS Kernal provides a simple interface to the hardware sprites supported by the C64. These routines control the sprites by writing to the VIC chip sprite registers as well as writing to the data space from which the VIC reads the sprite picture data. The reader should be familiar with the basic structure of sprite support on the c64 as explained in the Commodore 64 Programmer's Reference Guide. One of the space/function tradeoffs made in GEOS was to support only basic sprite functions. Applications requiring elaborate sprite manipulation, such as games, will probably not be using many of GEOS's features, whereas business, or text based applications will benefit from GEOS text, disk, and user interface features, and probably not need complicated sprite support.

The GEOS Kernal provides routines for drawing, erasing, and positioning:

- DrawSprite
- PosSprite
- EnablSprite
- DisablSprite

DrawSprite

Function: Draw/Redraw a sprite

Pass: r3L - sprite number to use (0 -7)
r4 - pointer to picture data

Return: 64 bytes copied from area pointed to by r4 to sprite picture data area for sprite whose number was passed in r4. (One extra byte is copied, even though sprites are only 63 bytes.)

Destroyed: a, y, r5

Synopsis: DrawSprite transfers the graphics data from the area pointed to by r4 to the RAM area that the VIC chip uses to display the sprite whose number was passed in r3L. Sprite 0 is used for the mouse/joystick cursor, and sprite 1 is used for the vertical bar text prompt cursor if promptOn is used.

PosSprite

Function: Place a sprite on the screen using GEOS (not hardware c64) x-y coordinates.

Pass: r3L - **sprite number** (0-7)
r4 - **x coordinate** (0-319)
r5L - **y coordinate** (0-199)

Return: r3L - unchanged

Destroyed: a, x, y, r6

Synopsis: PosSprite converts the straightforward GEOS coordinates for the indicated sprite, a two byte 0-319 coordinate for x and one byte 0-199 for y into the somewhat bizarre c64 hardware coordinate scheme. It stores the mutated values into the VIC hardware position registers, thus positioning the sprite.

EnablSprite

Function: Turns a sprite on so that it is visible on screen.

Pass: r3L - **sprite number** (0-7)

Return: VIC mobenbl register change to reflect sprite activated
r3L - unchanged

Destroyed: a, x

Synopsis: Flips the proper bit in the VIC mobenbl register for enabling the sprite whose number is passed in r3L.

DisablSprite

Function: Turns a sprite off.

Pass: r3L - **sprite number** (0-7)

Return: VIC mobenbl register changes to reflect sprite activated
r3L - unchanged

Destroyed: a, x

Synopsis: Flips the proper bit in the VIC mobenbl register for disabling the sprite whose number is passed in r3L.

Process Support

GEOS supports time-based processes. A process under the GEOS Kernal is a subroutine which is triggered to run every certain number of interrupts. The Kernal Interrupt code sets a flag when it is time for a process to run, and a MainLoop process dispatcher calls the process. Several Processes may be run simultaneously. Thus an alarm clock process could be going in one corner of the the screen, while the user is typing text into another, and a spreadsheet is recalculating itself in yet another. Another example of using processes is to check regularly for a condition to become true, and then when it does, take action. For example, changing the cursor picture from the arrow to an I-Beam when it moves from pointing at icons to pointing over text.

There are three parts to GEOS process support. GEOS supplies 1) an Interrupt-Level timer module to decrement the process timers, 2) a MainLoop dispatcher, to run process service routines when their timers run out, and 3) the application supplies a process definition table and service routine for the dispatcher to call.

When the process module is initialized, each process is assigned a timer and flag byte. Every 60th of a second the timer module at InterruptLevel decrements the process timers. When a timer reaches 0, its runnable bit is set in the process flag. GEOS MainLoop checks the runnable bit for each process and if it is set, the programmer designated service routine is called. The programmer provides a table containing the routines to call and how often to call them. An example is shown below:

ProcessTable:

```
.word    ProcessRoutine1
.word    N                                ;call ProcessRoutine1 every N interrupts
.word    ProcessRoutine2
.word    M                                ;call ProcessRoutine2 every M interrupts
```

That's all there is to it.

Processes may be runnable, blocked, or frozen. A runnable process is one whose timer is being decremented every interrupt and whose dispatch routine will be called each time it timers reaches zero. A frozen process is one whose timer is frozen and prevented from decrementing. It will thus never time out and not get run. A blocked process continues to have its timer decremented, but when it times out the routine is not run.

Where might one make use of these differences? Freezing a processes timer is probably most useful if you have two or more processes which you want to run in a certain order. If these need to be stopped for some reason, disabling interrupts, freezing them, and reenabling interrupts should maintain the relationship between their counters until they are unfrozen.

On the other hand, suppose you had a process which printed the value of of a clock every second but needed to be stopped from printing to the screen for some period of time. Blocking the process will allow its timer to run but makes sure that the time doesn't get written to the screen during a particular interval. The chances of the clock running slow because its missed a time out are less.

There is another process routine, called EnableProcess which forces a process's dispatch routine to be run once during the next MainLoop. If the process is blocked, or frozen

it will remain blocked or frozen. A running process will be executed once and begin counting down again.

GEOS also supports the concept of sleeping. This allows any routine, to halt execution for a specified time while allowing all other processing to take place. When the time interval is up, the routine "wakes up" and continues its execution. Note that in the meantime the values in registers may have changed.

It is unadvisable to have any routine which depends upon, or directly affects the state of the machine, or values of global variables, to use sleep because it is difficult to predict the state of the machine when the routines wakes up. Another caution is that when a routine wakes up, it wakes up from a call by the sleep manager in GEOS MainLoop. The routine is now running in a new environment. When sleep was first called, the sleep manager saved the address just after the jsr sleep in the routine. When the sleep time was up, the sleep manager did a jsr to the address it saved. When the routine finishes it returns not to its original caller, but to the sleep manager. In general, only routines called from MainLoop should sleep. If not, the routine will be waking up in an entirely different environment. Finally, it is generally a bad practice to have a routine sleep for some variable amount of time and then call another routine which may also sleep. There is the danger of creating a chain of events here that is difficult to debug. Sleep is good for printing messages to the screen that don't affect the state of the machine. For example, flashing one message, giving the user time to see it and then a few seconds later flashing another.

All process variables are hidden and are accessible only through the following process routines.

InitProcesses

Function: Initialize all processes, but don't start running them.

Pass: a - number of processes
r0 - pointer to process table

Return: r0 - unchanged

Destroyed: a, x, y

Synopsis: InitProcesses copies the process addresses and timer values from the given table and initializes each process as frozen and blocked. After initializing, start a process with RestartProcess.

RestartProcess

Function: Unblock and unfreeze a process and reset its timer.

Pass: x - the number of the process to start/restart

Return: x - unchanged

Destroyed: a

Synopsis: Starts a process running. First it unblocks and unFreezes the process so there is nothing to keep it from running. Then the process timer is reloaded with the value copied from the initialization table and the process begins running.

BlockProcess, UnblockProcess

Function: Block or Unblock a process dispatch routine from being run.

Pass: x - the number of the process to block/unblock

Return: x - unchanged

Destroyed: a

Synopsis: **BlockProcess:** Sets a flag so that the process's dispatch routine is prevented from being called. The process timer continues to count down and be restarted if the timer reaches zero before the process is unblocked. The process service routine is not called till it is unblocked.

UnblockProcess: Resets a flag so that the process's dispatch routine is may be called when the process times out.

FreezeProcess, UnfreezeProcess

Function: Freeze or unfreeze a process's timer.

Pass: x - the number of the process to freeze/unfreeze

Return: x - unchanged

Destroyed: a

Synopsis: **FreezeProcess:** Halt the indicated process's timer to prevent its dispatch routine from being run.

UnfreezeProcess: Restart the indicated process's timer, allowing it to timeout.

Sleep

Function: Pause execution for given time interval.

Pass: r0 - the amount of time to sleep for in sixtieth of a second

Return: undefined

Destroyed: undefined

Synopsis: Sleep remembers an address to be run at a later time and forces an rts. Thus if routine 1 calls routine 2 and routine 2 executes a sleep, a pointer to the code in routine 2 is saved and an rts that returns to routine 1 is performed. When the sleep times out, routine 2 will finish executing. The rts at the end of the routine will return to the sleep module since when execution was resumed, routine 2 was re-entered from the sleep module, not routine 1.

Sleep should be called either directly from an event-driven routine – so that the rts returns to Main Loop and the rest of the application continues to run – or be called from a single function subroutine like the following:

CallingRoutine:

```
some code
..
jsr PrintLater
...
rts
```

PrintLater:

```
lda #[N
;pass the number of interrupts to wait
sta r0L
lda #[N
sta r0H
jsr sleep
;execute the sleep and return to calling routine
...
Code to Print Some Thing
...
rts
```

In this case, PrintLater, a routine which doesn't depend upon or affect the state of the machine, is being dispatched to print something N interrupts later and the calling level routine is free to continue.

EnableProcess

Function: Force a process to be run.

Pass: x - the number of the process to have run

Return: x - unchanged

Destroyed: a

Synopsis: Forces the process's timer to be set to zero and for the process to be run once during the next Main Loop. If the process is blocked, or frozen it will remain blocked or frozen after its service routine is executed once. A running process will have its service routine executed once, its timer reset and resume counting down again.

EnableProcess is useful to make sure a process service routine is run once. The process might already be running, or it might be blocked or frozen. Enabling it will not change anything about the state of the process except to cause it to be run once.

EnableProcess is also useful if you are running your own interrupt code to support a special device, and from time to time you detect a condition that causes you to want to run a routine at MainLoop. The important distinction here is that the condition is flagged during Interrupt-Level, but the routine is run during MainLoop.

10.

Math Library

This section presents the math utility routines contained in the GEOS Kernal. These routines provide a variety of useful functions including word/byte multiplication and division. The routines are:

DShiftRight	Shift a word right n bits
BBMult	Byte by Byte multiplication
BMult	Byte by Word multiplication
DMult	Multiply two words
Ddiv	Divide two words
DSdiv	Divide two signed words
Dabs	Absolute value of word
Dnegate	Negate a twos-complement word
DDec	Decrement an unsigned word
GetRandom	Get a random word

DShiftLeft - Double Precision Shift Left

Function: Arithmetically shift operand left n bits. Computes **operand * 2ⁿ**

Called By: Utility

Pass: x - address of zpage pseudoregister (e.g., ldx #r1)
 y - the number of bits to shift left

Return: (x) - the register pointed to by x is shifted left y times
 a, x- unchanged

Destroyed: y - \$FF

Synopsis: DShiftLeft is double precision routine that arithmetically shifts left a 16 bit zpage register the number of bits passed in y. This equation for this computation is **operand * 2ⁿ**. All status flags are properly maintained.

DShiftRight - Double Precision Shift Right

Function: Arithmetically shift operand n bits right. Computes **operand** $\div 2^n$

Called By: Utility

Pass: x - address of a zpage pseudoregister (e.g., ldx #r1)
 y - the number of bits to shift right

Return: (x) - the register pointed to by x is shifted right y times
 a, x- unchanged

Destroyed: y - \$FF, all other registers unchanged

Synopsis: DShiftLeft is double precision routine that arithmetically shifts a 16 bit zpage register right the number of bits passed in y. This equation for this computation is **operand** $\div 2^n$. All status flags are properly maintained.

BBMult - Multiply Byte * Byte

Function: Multiply two unsigned byte operands and store product in word addressed by **x**.

Called By: Utility

Pass: x - address of the **destination** zpage pseudoregister (e.g., ldx #R1)
y - address of the **source** zpage pseudoregister (e.g., ldy #R2)

Return: (x) - the register pointed to by x gets the 16 bit result
x, y - unchanged

Destroyed: a, r7, r8

Synopsis: BBMult multiplies to bytes together and stores the word result in the destination. Thus if x contained the address of r5, the the low byte of r5 would be used as the first operand and the 16 bit product would be stored in both bytes of r5.

BMult - Multiply Word * Byte

Function: Multiply an unsigned word by an unsigned byte operand and store product in word indexed by x.

Called By: Utility

Pass: x - address of the **destination** zpage pseudoregister (e.g., ldx #R1)
y - address of the **source** zpage pseudoregister (e.g., ldy #R2)

Return: (x) - the register pointed to by x gets the 16 bit result
y, x - unchanged
(y) - high byte of the register pointed to by y zeroed

Destroyed: a, r6, r8

Synopsis: BMult multiplies a byte by a word and stores the result in the word. The byte source operand is pointed at by y. The destination word operand is pointed at by x. All status flags are properly maintained.

DMult - Double Precision Multiply

Function: Multiply two unsigned words.

Called By: Utility

Pass: x - address of the **destination** zpage pseudoregister (e.g., ldx #R1)
y - address of the **source** zpage pseudoregister (e.g., ldy #R2)

Return: (x) - the register pointed to by x gets the 16 bit result
x, y - unchanged
(y) - the register pointed to by y is left unchanged

Destroyed: a, r6 - r8

Synopsis: DMult multiplies the source word by the destination word and stores the result in the destination word. The word source operand is pointed at by y. The destination word operand is pointed at by x. All status flags are properly maintained.

Ddiv - Double Precision Divide

Function: Divide unsigned destination word by source word, and store quotient in destination. Store remainder in r8.

Pass: x - address of the **destination** zpage pseudoregister (e.g., rdx #r1)
y - address of the **source** zpage pseudoregister (e.g., ldy #R2)

Return: (x) - the register pointed to by x gets the 16 bit result
r8 - the remainder
x,y - unchanged
(y) - the register pointed to by y is left unchanged

Destroyed: a, r9

Synopsis: Ddiv divides a word destination by word source division and stores the result in the destination. The remainder is stored in r8. The word source operand is pointed at by y. The destination word operand is pointed at by x. All status flags are properly maintained. Thus for $4 \div 3$, the destination would get 1 and r8 would get 1.

DSdiv - Signed Double Precision Divide

Function: Divide signed source word by signed destination word. Store quotient in destination. Store remainder in r8.

Called By: Utility

Pass: x - address of the **destination** zpage pseudoregister (e.g., ldx #r1)
y - address of the **source** zpage pseudoregister (e.g., ldy #r2)

Return: (x) - the register pointed to by x gets the 16 bit result
r8 - the remainder
x, y - unchanged
(y) - the register pointed to by y is left unchanged

Destroyed: a, r9

Synopsis: DSdiv divides a signed destination word by signed source word and stores the result in the destination. The remainder is stored in r8. The word source operand is pointed at by y. The destination word operand is pointed at by x. All status flags are properly maintained. All arithmetic is done in two complement. Thus for $4 \div -3$, the destination would get -1 and r8 would get 1.

Dabs - Double Precision Absolute Value

Function: Compute the absolute value of a twos-complement word.

Called By: Utility

Pass: x - address of the zpage pseudoregister (e.g., ldx #r1)

Return: (x) - the register pointed to by x gets the twos-complement *absolute value* of its original contents

Destroyed: a

Synopsis: Dabs computes the absolute value of the twos-complement word stored in the register pointed at by x. All status flags are properly maintained. All arithmetic is done in twos-complement.

Dnegate - Signed Double Precision Negate

Function: Negate a twos-complement word.

Called By: Utility

Pass: x - address of the zpage word pseudoregister (e.g., ldx #r1)

Return: (x) - the word in the register pointed to by x gets negated
x - unchanged

Destroyed: a, y

Synopsis: Dnegate negates the twos-complement word stored in the register pointed at by x. All status flags are properly maintained. All arithmetic is done in twos-complement.

Ddec - Decrement an Unsigned Word

Function: Decrements an unsigned word.

Called By: Utility

Pass: x - address of the zpage word pseudoregister (e.g. ,ldx #R1)

Return: (x) - the unsigned word in the register pointed to by x gets decremented
x - unchanged

Destroyed: a

Synopsis: DDec decrements an unsigned word stored in the register pointed at by x. All status flags are properly maintained. All arithmetic is unsigned 16 bit.

GetRandom

Function: Get 16 bit pseudorandom number.

Called By: Utility

Pass: nothing

Return: random – contains a new 16 number.

Destroyed: a

Synopsis: GetRandom produces a new 16 bit pseudorandom number in the global variable random. $\text{new} = (\text{old} + 1) * 2 \bmod 65521$. The random number returned in random is always less than 65221, and has a more or less even distribution between 0 and 65221.

CopyFString - Copy Fixed Length String

Function: Copy a given number of bytes from one memory area to another

Called By: Utility

Pass: x - address of zpage register containing pointer to **source string**
y - address of zpage register containing pointer to **destination string**
a - the number of bytes to copy

Return: ((x)) ->((y)) - string copied from area beginning at location pointed to register indicated by x to location indicated by register pointed to by y

Destroyed: a, x, y

Synopsis: CopyFString copies a fixed length null terminated byte string from one memory area to another.

CmpString

Function: Compare a null terminated string at one memory area to another.

Called By: Utility

Pass: x - address of zpage register containing pointer to **source string**
y - address of zpage register containing pointer to
destination string

Return: zero flag - set if strings were equal.
minus flag - set if the first source byte was smaller than the destination
byte for the first unequal pair of bytes.

Destroyed: a, x, y

Synopsis: CmpString compares an arbitrary length null terminated byte string at one memory area to the string at another. An unsigned **cmp** is done so if the strings are unequal then the minus bit in the status byte is set if the byte in the source string which didn't match the corresponding byte in the destination was the smaller of the two, i.e., source < dest.

11.

General Library Routines

These are general routines for supporting string manipulation, data moving, initialization, etc. These are the same routines the GEOS Kernal uses. These routines include:

CopyString	Copies one null terminated string to another
CopyFString	Copies a fixed length string
CmpString	Compares one null terminated string with another
CmpFString	Compares two fixed length strings
Panic	What happens when a BRK instruction is executed
MoveData	Moves arbitrarily sized memory area to another area
ClearRam	Clears any sized section of memory
FillRam	Fills any sized section of memory with a specific byte
InitRam	Reads a table to initialize a section of memory
ToBasic	Passes control from GEOS to BASIC
CallRoutine	Does the routine whose address is passed

FirstInit	Initializes the entire system
GetSerialNumber	Returns the Kernal serial number
CRC	Computes a checksum
ChangeDiskDevice	Changes device number of current disk on the serial bus

CopyString

Function: Copy null terminated string from one memory area to another

Called By: Utility

Pass: x - address of zpage register containing pointer to **source string**
y - address of zpage register containing pointer to **destination string**

Return: ((y)) - string copied to memory location pointed to by register pointed to by y

Destroyed: a, x, y,

Synopsis: CopyString copies an arbitrary length null terminated byte string from one memory area to another.

CmpFString – Compare Fixed Length String

Function: Compare a given number of bytes at one memory area to another

Called By: Utility

Pass: x - address of zpage register containing pointer to **source string**
y - address of zpage register containing pointer to **destination string**
a- the number of bytes to compare

Return: zero flag - zero flag set if strings were equal
minus flag - set if source byte which didn't match destination byte was smaller than it.

Destroyed: a, x, y,

Synopsis: CmpFString compares a fixed length byte string from one memory area to another. Since the number of bytes to compare is passed in a maximum of 256 bytes may be compared. Unlike with CmpString which stops at a null byte in the source string, the strings may contain zero bytes. An unsigned **cmp** is done so if the strings are unequal then the minus bit in the status byte is set if the byte in the source string which didn't match the corresponding byte in the destination was the smaller of the two, i.e., source < dest.

Panic – Roll Over and Die

Function: Standard way of trapping a BRK instruction. Puts up a dialog box with address of the BRK.

Called By: usually inadvertently

Calls: DoDialog

Pass: nothing

Return: nothing

Destroyed: a, x, y, r0 - r15, and probably much, much more

Synopsis: When a BRK instruction is encountered GEOS traps the interrupt and prints out a Dialog Box containing the address of the BRK instruction. This is useful for debugging code which branches off into space and wanders about until it eventually tries to execute a 0 instruction. Panic prints out a dialog box that contains the message "System error near #xxxx," where xxxx is the address of the BRK instruction encountered. Panic ceases operation of the machine.

MoveData, i_MoveData

Function: Move a large block of data in memory

Called By: Utility

Pass: r0 - **source** address
r1 - **destination** address
r2 - **number of bytes** to move

Inline: .word - **source** address
.word - **destination** address
.word - **number of bytes** to move

Return: The data is moved.

Destroyed: a, y, r0, r1, r2

Synopsis: MoveData moves a block of data of any length from one area in memory to another. The source and destination areas may overlap.

ClearRam

Function: Zero out a section of memory.

Called By: Utility

Pass: r0 - number of bytes to clear
r1 - address to start clearing memory

Return: nothing

Destroyed: a, y, r0, r1, r2L

Synopsis: Clear a section of memory by setting all bytes within to 0.

FillRam, i_FillRam

Function: Fill the bytes in a selected area of memory with a given byte.

Called By: Utility

Pass: r0 - **number of bytes** to clear
 r1 - **address of first byte** to set
 r2L - **the value** to store in each byte in the selected area of RAM

Inline: .word - **number of bytes** to clear
 .word - **address to start** clearing memory
 .byte - **the byte** to store in memory area

Return: nothing

Destroyed: a, y, r0, r1, r2L

Synopsis: Set all the bytes in the indicated area of memory with the given byte. This routine is useful for setting an area in memory with a value other than zero, typically \$FF or one of the pattern bytes (see patterns in the Appendix).

InitRam

Function: Provides simple yet powerful way to initialize areas of memory

Called By: Utility

Pass: r0 - address of Initialization Table as specified below

Return: Area(s) of memory initialized.

Destroyed: a, x, y, r0 - r1

Synopsis: InitRam reads a table specifying how to initialize a block of RAM. InitRam is especially useful used in two ways:

1. A quick, compact way of performing an initialization which must be done several times to an odd length area of RAM. Resetting an application to a default state is a typical use.
2. Initializing a bunch of noncontiguous small segments of RAM. InitRam is especially efficient in performing an initialization in a "two bytes here, three bytes there" fashion.

The instruction table, pointed to by r0, contains one or more entries, each of the form:

```
.word Location          ;The first word specifies the location of the
                        ; RAM to be initialized.
.byte number of bytes
.byte value1, value2, ... value n
```

There may be several of these entries in the initialization table. A word with value of 0 where the beginning of an entry is expected terminates the table. An example of a two entry table follows.

Sample Initialization Table

.word	FirstLocation	; Specifies the location of the ;RAM to be initialized.
.byte	3	;Three bytes to initialize here
.byte	4,5,6	;Values to store in the three bytes ;starting at FirstLocation
.word	SecondLocation	;Second location to be initialized.
.byte	10	;Ten bytes to initialize here
.word	30,\$FFE5	;we can mix words and bytes, just
.byte	0,0,23,1	; so there are only 10 bytes in total
.word	0	
.word	0	;a word with value 0 where an address ;is expected ends the table

CallRoutine

Function: Perform an indirect jsr to the routine whose address is stored in **x** and **a**.

Pass: **a** - **low byte** of routine to call
 x - **high byte** of routine to call

Return: Depends on routine called; returns nothing itself.

Destroyed: Depends on routine called; destroys nothing itself.

Synopsis: CallRoutine performs a pseudo call indirect to the address passed in **a** and **x**. If **a** and **x** are 0, no routine is called.

GetSerialNumber

Function: Returns the serial number from the GEOS Kernal.

Pass: nothing

Return: r0 - The serial number

Destroyed: a

Synopsis: GetSerialNumber returns the GEOS serial number. Upon the first time it boots, GEOS will generate a unique serial number. This value is stored in the Kernal. Applications may key themselves to this serial number and not allow themselves to be run from a GEOS Kernal that has a different one. The application will then only run on the system of the person who bought. Other user's GEOS Kernals will have different serial numbers.

ToBasic

Function: Passes control from GEOS to BASIC. A BASIC command may be specified to be immediately run.

Pass: r7 - New start of variable space pointer for BASIC command
Text - NULL-terminated command string for BASIC

Return: Doesn't return.

Destroyed: a

Synopsis: ToBasic is used to transfer control to BASIC. The c64 Kernal, I/O space and BASIC ROMS are bank switched in and the system reinitialized.

FirstInit

Function: FirstInit does a complete warm start of the system and then jumps to the application's init code whose address is passed in r7.

Pass: nothing

Return: The system in its warm start configuration.

Destroyed: a, y, r0, r1, r2L

Synopsis: FirstInit is the routine called to initialize the system and then run an application whose address is passed in r7. The WarmStart table in Chapter 19 contains a listing of all the memory locations FirstInit affects.

CRC

Function: CRC performs a checksum on specified data

Pass: r0 - pointer to start of data
r1 - # of bytes to check

Return: r2 - checksum.

Destroyed: a, x, y, r0, r1, r3L

Synopsis: CRC does a long and involved checksum computation on the memory area passed.

ChangeDiskDevice

Function: Tells the current disk drive to change its serial bus number

Pass: a - new device number; 8, 9, 10, or 11

Return: x - disk error status, 0 = OK. See disk errors in the Appendix

Destroyed: a, x, r1

Synopsis: Change DiskDevice is called by the deskTop when doing an add drive. This routine sends a new serial bus device number to the disk.



Dialog Boxes

Dialog Boxes appear as a rectangle in which text, icons, and string manipulation may occur. Dialog Boxes are used by applications to display error conditions, warn the user about possibly unexpected side effects, prompt for a sentence or two of input, present filenames for selection, and perform various other tasks where user participation is desired. Several frequently used Dialog Box functions are built directly into the the GEOS Kernal. Along with programmer defined functions, Dialog Boxes provide a simple, compact, yet flexible user interface.

A Dialog Box may be called up on the screen at any time. It is like a small application, running in its own environment. It will not harm the current application, or change any of its data (unless this is intentionally done by a programmer supplied routine). Calling up a Dialog Box causes most of the state of the machine to be saved. All the Kernal variables, vectors, and menu and icon structures are saved. The Dialog Box can therefore be very elaborate, since it need not worry about permanently affecting the state of the machine. The

pseudoregisters r0H - r15, however, are not saved, nor is the screen under where the Dialog Box appears. Restoring the screen appearance after a DB is run is described later.

To call up a Dialog Box use the routine DoDlgBox. To exit from a Dialog Box and return to the application call RstrFrmDialog. All the variability of Dialog Boxes is provided by a powerful yet simple table. The table specifies the dimensions and functionality of the Dialog Box (DB). DB tables are made up of a series of command and data bytes. DB command bytes indicate icons to display or commands (usually for printing text) to execute within the DB. DB data bytes specify information such as location of the DB, its dimensions, and text strings.

DB Icons and Commands

The Kernal supports a special set of resident icons for use in DBs. DB Icons provide a simple user response to a question or statement. When the user clicks on one of these icons the DB is erased, the number of the selected icon is returned in r0L, and RstrFrmDialog is automatically called. The application that called DoDlgBox then checks r0L and acts accordingly, usually calling a routine it associates with that icon. DB Icons indicating YES, NO, OK, OPEN and DISK are provided.

DB Commands are provided for running any arbitrary routine, printing a text string, prompting for and receiving a text string, putting up a scrolling filename box, putting up a user-defined icon, and providing a routine vector to jump through if the joystick button is pressed when the cursor is not over any icon. DB Commands take the form of one command byte containing the number of the command to execute and any following optional data bytes.

DB Structure

The first entry in a DB table is a command byte defining its position. This can either be a byte indicating a default position for the DB, DEF_DB_POS, or a byte indicating a user defined position, SET_DB_POS which must be followed by the position information.

The position command byte is or'ed with a system pattern number to be used to fill in a shadow box. The shadow box is a rectangle of the same dimensions as the DB and is filled with one of the system patterns. The shadow box appears underneath the DB one can

to the right and one card below. A system pattern of 0 indicates no shadow box. It's easier to look at an example of DB with a shadow box than it is to describe it. A picture of one appears in the Open Box example later in this chapter.

The two forms for the position byte, default and user defined, are:

<code>.byte</code>	<code>DEF_DB_POS</code>	<code> pattern</code>	<code>.byte</code>	<code>SET_DB_POS</code>	<code> pattern</code>
			<code>.byte</code>	<code>top</code>	<code>;(0-199)</code>
			<code>.byte</code>	<code>bottom</code>	<code>;(0-199)</code>
			<code>.word</code>	<code>leftside</code>	<code>;(0-319)</code>
			<code>.word</code>	<code>right</code>	<code>;(0-319)</code>

Position Command

After the position byte (or bytes) may appear a number of icon or command bytes. The OK icon is the most common icon. The OK byte is followed by two bytes defining the position of the icon as an offset from the upper left corner of the DB. The first is the x position in bytes, 0-319; the second is the y position in pixels, 0-199. The OK icon would look like the following:

<code>.byte</code>	<code>OK</code>
<code>.byte</code>	<code>xoffset</code>
<code>.byte</code>	<code>yoffset</code>

OK Icon Byte

Whenever a system DB icon is activated, the DB exits, returning the icon's number in r0L. The application can then know which icon was selected and take the appropriate action. A maximum of 8 icons may be defined in a DB.

The following table contains the icon commands.

Dialog Box Icons		
Icon	Value	Example
OK	1	.byte OK .byte xpos_in_bytes .byte ypos_in_lines
Cancel	2	.byte Cancel .byte xpos_in_bytes .byte ypos_in_lines
Yes	3	.byte YES .byte xpos_in_bytes .byte ypos_in_lines
NO	4	.byte NO .byte xpos_in_bytes .byte ypos_in_lines
OPEN	5	.byte OPEN .byte xpos_in_bytes .byte ypos_in_lines
DISK	6	.byte DISK .byte xpos_in_bytes .byte ypos_in_lines
FUTURE1	7	.byte FUTURE1 .byte xpos_in_bytes .byte ypos_in_lines
FUTURE2	8	.byte FUTURE2 .byte xpos_in_bytes .byte ypos_in_lines
FUTURE3	9	.byte FUTURE3 .byte xpos_in_bytes .byte ypos_in_lines
FUTURE4	10	.byte FUTURE3 .byte xpos_in_bytes .byte ypos_in_lines

Dialog Box Commands

Several commands are defined for use in DBs. Many are used to put up text within the Box. For example, the command DBTXTSTR (=11) is followed by two position offset

bytes and a word pointing to a text string. When used in a DB, DBTXTSTR will display the text string at a position offset from the upper left corner of the DB. The position offsets are measured in pixels from top of the DB to the baseline of the text string, and in pixels from the left side of the DB to the left side of the first character in the string. This means any string may be offset at most 256 pixels from the left side of the DB. The following table contains the available commands.

Dialog Box Commands

Command	Value	Command Name	
Example		Description	
DBTXTSTR	11	Dialog Box Text String	
.byte DBTXTSTR .byte xpos_offset .byte ypos_offset .word TextPtr	;command byte ;offset from left of DB in pixels 0-256 ;offset from top of DB in pixels 0-199 ;pointer to null terminated string.	Display the text string pointed to by TextPtr positioned xpos_offset pixels from the left and ypos_offset pixels from the top of the DB. TextPtr contains address of null terminated string.	
DBVARSTR	12	Dialog Box Variable Text String	
.byte DBVARSTR .byte xpos_offset .byte ypos_offset .byte RegNumber	;command byte ;offset from left of DB in pixels 0-256 ;offset from top of DB in pixels 0-199 ;r5 - r10, r14. num of Reg.	Display the text string pointed to by TextPtr at a position starting xpos_offset pixels from the left of the DB and ypos_offset pixels from the top. Load Reg with pointer to null terminated string.	
DBGETSTRING	13	Dialog Box Get Text String	
.byte DBGETSTRING .byte xpos_offset .byte ypos_offset .byte RegNumber .byte MaxChars	;command byte ;display user typed text at this x offset ;in pixels from the left side of the DB ;offset from top of DB in pixels ;pass 5 - 10 for r5 - r10 ;maximum number of chars to accept	Read a text string typed by user into the buffer pointed to by RegNumber, ie. If RegNumber is 5 then pointer is in r5. Display this string inside the DB at x offset/y offset from the upper left corner. MaxChars is the maximum number of chars to read into the input buffer.	
DBSYSOPV	14	Dialog Box System Other Press Vector	
.byte DBSYSOPV	;command byte	Enable function that causes a return to the application whenever mouse is pressed any place except over an icon.	
DBGETFILES	15	Dialog Box Get Files	
.byte DBGETFILES .byte xpos_offset .byte ypos_offset r7L: file type r5: filename buffer r10: ptr to permnt name	;command byte ;offset from left of DB in bytes 0-39 ;offset from top of DB in pixels 0-199 ;GEOS file type. ;pointer to buffer to return filename in ;restricts filenames displayed	Display the filename box inside the DB. The box is x wide, y tall. The filename box displays a list of filenames. Any filename can be selected by the user. It is then copied into the buffer pointed to by r5. If r10 is not zero, it points to a string which contains the permanent name string, e.g., "Paint Image" taken from the File Header. In this case, only geoPaint documents will be displayed for selection. If there are more files than can be displayed within the box, pressing the scroll arrows that appear under the filename box will scroll the filenames up or down. Max of 16 files supported.	

Dialog Box Commands

DBOPVEC	17	Dialog Box User Other Press Vector
.byte DBOPVEC ;command byte .word UserVector ;User defined other press vector.		Enable function that causes a jsr to the routine pointed at by the word following the command byte whenever the mouse is pressed any place except over an icon or menu. This routine may return to the DB code or it may perform a jmp to RstrFrmDialog to return to the application.
DBUSRICON	18	Dialog Box User Icon
.byte DBUSRICON ;command byte .byte x_offset ;x position offset for icon in bytes .byte y_offset ;y position offset for icon .word User_Icon ;pointer icon info table Icon Info Table: .word iconPic ;pointer to graphics data bytes .byte 0 ;x position, (already set above) .byte 0 ;y position, (already set above) .byte width_bytes ;6 bytes wide is default icon width .byte height_pixels ;16 pixels high is default height .word Service_Routine ;routine to call when icon is activated		Place and execute a user defined icon. The icon info table includes pointers to a service routine and the icon picture. When this icon is activated, a jsr to the routine is done. This routine may jmp to RstrFrmDialog or another system icon may be provided for exiting from the DB.
DB_USR_ROUT	19	Dialog Box User Routine
.byte DB_USR_ROUT;command byte .word UserVector ;pinter to user defined routine		This is the do-it-yourself DB function. The user routine pointed to by UserVector is called when the DB is first drawn. This routine can be used to do just about anything with the DB. Its rts returns to the DB code unless an alternate exit is provided.

The registers r5 through r10 and r15 may be used to pass parameters to those commands expecting them. A couple of the commands deserve further explanation.

DBOPVEC

DBOPVEC sets up a vector which contains the address of a routine to call whenever the user clicks outside of an icon. This routine will be run and its rts will return to the DB code in MainLoop. Other icons or DB commands may then be executed, or icons selected.

If the programmer wants the routine to exit from the DB altogether as DBSYSOPV does, then a `jmp RstrFrmDialog` should be executed from within the routine. Whenever this is done, `sysDBData` should be loaded with a value that `RstrFrmDialog` will then transfer to `r0L` when it exits. In situations where several user responses are possible within a DB, the calling application checks `r0L` to determine the action that caused the DB exit. Your DBOPVEC routine should return `sysDBData` a value that cannot be mistaken for a different icon in the same DB. Since DBs can only handle 8 icons, any number greater than 8 is sufficient.

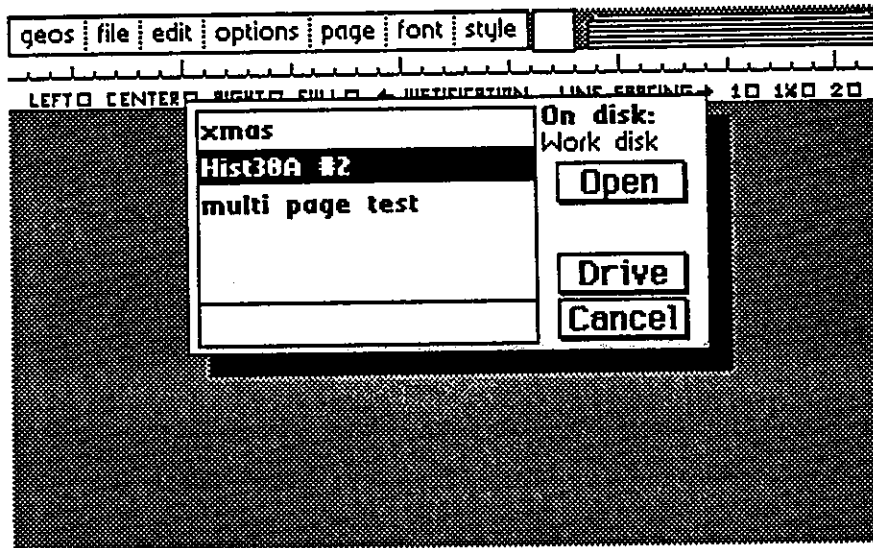
DBUSRICON

If the programmer wishes to have an icon in a DB that is not one of the Kernal supported DB icons, he may use the DBUSRICON command to define his own. A word following the command byte points to an icon table not unlike the table normally used to define icons within an application. As can be seen in the Dialog Box Commands diagram above, the position bytes for the icon within this table are set to zero as the position offset bytes just following the command byte are used instead. The user routine pointed to from inside this icon table is executed immediately when a press within the icon is detected. Like DBOPVEC, instead of returning to the application like the predefined system icons, this user icon returns to the DB level in MainLoop.

To make the user routine return to the application it may execute a `jmp RstrFrmDialog`. A QUIT or OK icon may also be used in the same DB to cause a return to the application. As with DBOPVEC, the DBUSRICON routine should load `sysDBData` with a value that `RstrFrmDialog` will then transfer to `r0L`. This value should be selected so that the application will not mistake it for one of the DB icons.

DBGETFILES

The DBGETFILES DB command is the most powerful. A picture of it appears below.



A box containing the names of files which can be selected is displayed. If there are more files than can be displayed at one time, the up/down arrow icon can be used to scroll the filenames up or down. A maximum of 15 files may be viewed this way. Usually this is enough. Upon execution of the DB, r7L is expected to contain the GEOS file type (SYSTEM, DESK_ACC, APPLICATION, APPL_DATA, FONT, PRINTER, INPUT_DEVICE, DISK_DEVICE). r5 should point at a buffer to contain the selected filename. If the caller passes a filename in r5 and this file is one of the files found by DBGETFILES, then this filename will appear highlighted when the filenames are displayed in the dialog box.

When a file is selected, its name will be null terminated and placed in this buffer. r10 should be set to null to match all files of the given type, or point to a buffer containing the permanent name string of files to be matched. The permanent name string is contained in the File Header block for each file. It contains a name that is the same for all files of the same type. For example, geoPaint will only want to open files it created. It points r10 to the string "Paint Image", when using DBGETFILES. This is useful for displaying only those files of GEOS type APPL_DATA created by a specific program.

The end of a DB definition table is signalled with a .byte 0 as the last entry. As examples speak louder than explanations we present two DB examples below.

DB_USR_ROUT

The DB_USR_ROUT command executes a programmer supplied routine when the DB is drawn. This routine may be quite elaborate, setting up processes, menus, edit windows and the like. Since DoDlgBox and RstrFrmDialog, respectively, save and restore the system state, a DB_USR_ROUT called routine need not worry about trashing the state of the system. However, you may not call DoIcons from within a DB_USR_ROUT if you are also using the standard Dialog Box Icons as the two sets of icons will interfere. The DB icon structure is drawn and initialized after the DB_USR_ROUT is called. This way an icon may be placed on top of a graphic drawn by the DB_USR_ROUT.

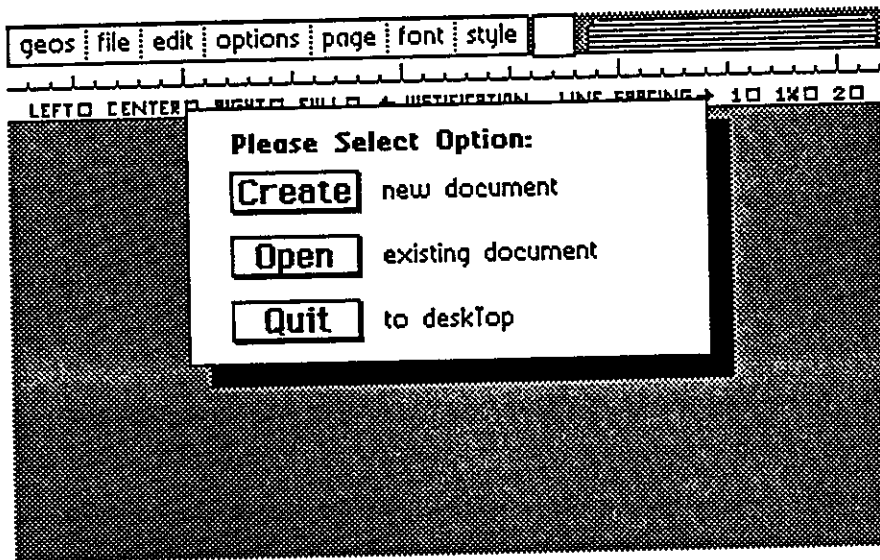
Exiting from a DB

The applications screen is recovered in one of two ways. First, if the screen's contents are buffered to the background screen, then all that needs to be done is a RecoverRectangle which will copy the background screen to the foreground screen. If the DisplayBufferOn flag is set so that the background is being used for code space and not to buffer the foreground screen, then the application must provide another means to recreate the screen appearance.

When RstrFrmDialog is called it will call the routine whose address is in RecoverVector. RecoverVector normally contains the address of RecoverRectangle. To recover the screen when the display is being buffered to calls through RecoverVector are done. First, the RstrFrmDialog routine sets up the coordinates the DB's shadow box and vectors through RecoverVector. This will restore the area under the shadow box. Second, it sets up the coordinates of the area under the DB itself and vectors through RecoverVector again. In this way the contents of the Background Screen corresponding to the area under the DB and its shadow box are copied to the Foreground Screen.

If the application does not use the Background Screen RAM as a screen buffer then it must provide the address of a different routine to call. An alternate routine address may be stored in RecoverVector in order to provide some other means of recreating the screen appearance. The dimensions of the areas to recreate are passed in the regular RecoverRectangle registers r2 - r4. This other routine will be called twice and may use the dimensions of the area to recover as passed in r2 - r4. If this routine need only be called once, then it should set a flag for itself so that the second time it gets called it can simply return.

DB EXAMPLE: THE OPEN BOX STAT



Dialog Box Example: openBox

Description: A table from geoWrite for putting up Dialog Box for selecting a new document, opening an existing document, or quitting geoWrite altogether.

openBox:

```
.byte DEF_DB_POS|1    ; standard DB with shadow pattern #1

.byte DBTXTSTR        ; display the select string
.byte TXT_LN_X        ; place it at the standard x offset (=16 pixels)
.byte 2*8             ; y offset in pixels from top of box
.word selectOptionTxt; pointer to the message "Please Select Option:"

.byte DBUSRICON       ; programmer defined NEW icon
.byte 2               ; x offset in bytes for left side of icon
.byte 3*8             ; y offset in pixels for top of icon
.word newIcon         ; pointer to the icon table for the Create icon

.byte DBTXTSTR        ; put up "new document" text
.byte (2+6)*8+7       ; x offset: (2 + width of icon) * 8 pixels/byte + 7
.byte 3 * 8+10        ; y offset: 10 below top of Create icon
.word newOptionTxt    ; pointer to text for "new document"

.byte OPEN            ; standard system OPEN icon
.byte 2               ; x offset in bytes
.byte 6 * 8           ; y offset, 24 pixels below Create icon

.byte DBTXTSTR        ; put "existing document" to the right of the icon
.byte (2 + 6) * 8 + 7; x offset (2 + width of icon) * 8 pixels/byte + 7
.byte 6 * 8 + 10      ; y offset for string, 24 below "new document"
.word openOptionTxt   ; pointer to "existing document"

.byte DBUSRICON       ; programmer defined NEW icon
.byte 2               ; x offset in bytes for left side of icon
.byte 9*8             ; y offset in pixels for top of icon
.word quitIcon        ; pointer to the icon table for the Quit icon

.byte DBTXTSTR        ; put up "to deskTop" text
.byte (2+6)*8+7       ; x offset: (2 + width of icon) * 8 pixels/byte + 7
.byte 9 * 8+10        ; y offset 24 below top of Open icon above
.word quitOptionTxt   ; pointer to text for "to deskTop"
.byte 0               ; signals end of DB
```

```

selectOptionTxt:      ; the select option message with embedded
                      ; BOLDON and and PLAINTEXT bytes to turn
                      ; boldface on and off.
        .byte BOLDON,"Please Select Option:",PLAINTEXT,0
newOptionTxt:
        .byte "new document",0
                      ; note each of these strings are null terminated
openOptionTxt:
        .byte "existing document",0
quitOptionTxt:
        .byte "to deskTop",0
newIcon:              ; icon definition table
        .word createIconPic ; address of picture data for the Create icon
        .byte 0             ; x position, ignored here, use offset in DB table
        .byte 0             ; y position, ignored here.byte
        .byte SYS_DB_ICN_WIDTH
                      ; uses standard width of a system icon (6 bytes)
        .byte SYS_DB_ICN_HEIGHT
                      ; standard height of system icon (16 pixels)
        .word createIconServ ; pointer to the service routine which creates the
                      ; file, and returns to the application
quitIcon:              ; icon definition table
        .word quitIconPic   ; address of picture data for the Quit icon
        .byte 0             ; x position, ignored here, use offset in DB table
        .byte 0             ; y position, ignored here.byte
        .byte SYS_DB_ICN_WIDTH
                      ; uses standard width of a system icon (6 bytes)
        .byte SYS_DB_ICN_HEIGHT
                      ; standard height of system icon (16 pixels)
        .word createIconServ ; pointer to the service routine which quits to the
                      ; deskTop
createIconServ:        ; the simple service routine for the create icon
        lda    #1         ; indicate icon number as if OK icon was
        bne    jmpIconServ ; activated.
quitIconServ:
        lda    #2         ; return value for quit
jmpIconServ:
        sta    sysDBData   ; store icon number before RstrFrmDialog call
        jmp    RstrFrmDialog ; exit from DB
createIconPic:
        96 bytes of graphics data for icon
quitIconPic:
        96 bytes of graphics data for icon

```

The following DB example is for the GetFiles DB.

Dialog Box Example: LoadBox/GetFiles

Description: A DB table from geoWrite for putting up Dialog Box for loading an existing document. If Open was selected from the openBox DB above, then the GetFiles DB is displayed to help the user choose from the available files.

Note:

r7L: contains GEOS file type for the files to select

r5: points to the buffer to selected store filename

r10: is 0 or points to permanent name string, in this case "geoWrite Doc".

LoadBox:

```
.byte DEF_DB_POS|1          ; standard DB, with shadow in pattern #1. The
                             ; GetFiles box works well inside a standard DB
.byte DBTXTSTR              ; display string, "current disk:"
.byte DB_ICN_X_2 * 8 - 6    ; x offset in pixels for text. (=17*8-6 = 130)
.byte TXT_LN_1_Y - 6        ; y offset
.word curDiskString         ; ptr to "current disk:"string above disk name

.byte DBTXTSTR              ; string holding the name of the current disk
.byte DB_ICN_X_2 * 8 - 6    ; x offset in pixels for text. (=17*8-6 = 130)
.byte TXT_LN_2_Y - 12       ; y offset
.word diskName              ; ptr to buffer already loaded with disk name

.byte DBGETFILES            ; the filename box command
.byte 4                     ; x offset in pixels from left side of DB
.byte 4                     ; y offset in pixels from top of DB
.byte OPEN                  ; use the Open DB
.byte DB_ICN_X_2            ; a standard x offset for Open
.byte 24                    ; y offset from top of DB
.byte CANCEL                ; use standard CANCEL icon
.byte DB_ICN_X_2            ; x offset for CANCEL
.byte 72                    ; y offset for CANCEL

.byte DISK                  ; Disk icon, when two drives are connected
                             ; service routine opens other disk drive
.byte DB_ICN_X_2            ; x offset for Disk
.byte 48                    ; y offset for icon
.byte 0                     ; end of DB definition
```

curDiskString:

```
.byte BOLDON."On disk:", PLAINTEXT,0 ;the string to print in the DB
```

The Dialog Box Routines

Following are the routine descriptions for all the callable routines for processing DBs.

DoDlgBox

Function: Display a Dialog Box

Called By: Applications Programmer

Pass: r0 - pointer to Dialog Box definition table

Return: Returns through RstrFrmDialog
r0L - returned with a value indicating the action that caused the DB to exit. If a system icon was selected, r0L will contain its number. DoDlgBox doesn't actually return to the application. The return is done through RstrFrmDialog.

Destroyed: a, x, y, r0 - r4, stack is preserved, as are all buffers and global RAM.

Synopsis: DoDlgBox invokes a Dialog Box. The caller passes the address of the DB definition table. All system RAM variables, (disk, menu, icon, process, etc) except for r0 - r4 are saved to be later restored by RstrFrmDialog. User routines may provide processes, menus, icons without fear of trashing the state of the application upon return. The registers r5 - r10 may be used for passing information from the application to the DB.

RstrFrmDialog

Function: Performs an exit from a Dialog Box routine

Called By: Programmer from his DB service routines and system for normal DB exit.

Pass: sysDBData – an icon number to return in r0L. This indicates the source of the return. This is not mandatory unless the use of the dialog box necessitates the ability to distinguish between two or more possible return sources, e.g., both YES and NO system icons are used.

Return: r0L - contents of sysDBData is transferred to r0L, usually contains the number of an icon if one was selected.

Other - since programmer supplied routines are callable from within DB processing, an wide range of values may be returned.

System RAM - except for the registers, all system global RAM (menus, icons, buffers) are restored.

Destroyed: a, x, y, r0H - r14

Synopsis: This routine is called from system DB icon routines as well as from programmer supplied service routines. See the discussion above. For example, if the DB definition table provides for an OK icon and this icon is selected, then the system routine for OK will load the number of the OK icon into sysDBData and jmp to RstrFrmDialog. r0L will be loaded with the contents of sysDBData and the system RAM variables are restored (they were saved by DoDlgBox) except for the registers r0 - r1.

File System

The GEOS file system is based on the normal c64 DOS file system. A combination of two factors led to an augmentation of the basic structure: first, the c64 was not originally designed to be a disk computer, and second, the addition of the diskTurbo now makes it practical to read and write parts of a file as needed. Previously the slowness of the disk drive often meant that files were read in at the beginning of execution, and not written until exiting the program. If file writes had to be done in the middle of execution, a coffee break was usually warranted.

GEOS supports two different types of files. The first is similar to regular c64 files and is called a SEQUENTIAL* file. This type of file is made up of a chain of sectors on the disk. The first two bytes of each sector contain a track and sector pointer to the next sector on the disk, except for the last sector which contains \$00 in the first byte to indicate that it is the last block, and an index to the last valid data byte in the sector in the second byte. The

* SEQUENTIAL stands for any non-VLIR file in GEOS, and should not be confused with the SEQ c64 file format. In fact USR, PRG and SEQ c64 files are all qualify as GEOS SEQUENTIAL file types.

second type of file is a new structure, called a Variable Length Indexed Record, or VLIR for short. An additional block, called a Header Block, is added to both VLIR SEQUENTIAL files. It contains an icon graphic for the file, as well as other data as discussed later.

To understand GEOS files, one must first understand the Commodore files on which they are based. I refer the reader to any of the several good disk drive books available. I use the Commodore 1541 (or 1571) User's Guide, and The Anatomy of the 1541 Disk Drive (from Abacus Software).

This chapter is divided into three sections. The first, for those already familiar with the 1541, is a brief refresher of the basic Commodore DOS. Second we present GEOS routines for opening and closing disks and dealing with directories and standard files. The final section is devoted to a detailed look at VLIR files.

The Foundation

A c64 disk is divided into 35 tracks. Each track is a narrow band around the disk. Track 1 is at the edge of the disk and track 35 is at the center. Each track is divided into sectors, which are also called blocks. The tracks near the outside edge of the disk are longer and therefore can contain more blocks than those near the center. The Block Distribution by Track table shows the number of sectors in each track.

Block Distribution by Track		
Track Number	Range of Sectors	Total Sectors
1 to 17	0 to 20	21
18 to 24	0 to 18	19
25 to 30	0 to 17	18
31 to 35	0 to 16	17

Track 18, the directory track, is used to hold information about the individual files contained on the disk. Sector 0 on this track contains the Block Availability Map (BAM) and the Directory Header. The BAM contains 1 bit for every available block on the disk. The bits corresponding to blocks already allocated to files are set while the bits corresponding to free blocks are cleared. Before the BAM bits is a pointer to the first Directory Block, which is described later. The BAM format is unchanged by GEOS.

The Directory Header contains the disk name, an ID word (to tell different disks apart), and three new elements for GEOS, a GEOS ID string, a track/sector pointer to the Off Page Directory block, and a disk protection byte. The GEOS ID string is contained in an otherwise unused portion of the BAM/Directory Header Block. It identifies the disk as a GEOS disk and identifies the version number, which can be important for data compatability between present and future versions of GEOS. See the BAM Format/Directory Header table below. This string should not be confused with the GEOS Kernal ID and version string at \$C000 as described in chapter 1.

The Off Page Directory Block is a new GEOS structure but has the same format as regular c64 Directory Blocks. Directory Blocks hold up to 8 Directory Entries. Each Directory Entry (also known as File Entry because it describes a file), contains information about one file. When a file is moved off the deskTop notepad onto the boarder, the file's Directory Entry is erased from its Directory Block and is copied to the Off Page Directory Block. A buffer in memory is also reserved to save information about each file on the boarder. The Off Page feature exists so that a file can be copied between disks on a one drive system. The Icon for an off page file will remain on the deskTop border when a new disk is opened and the deskTop set to display the contents of the new disk. The file can then be dragged to the notepad from the boarder, thus copying it to the new disk.

The disk protection byte is at byte 189 = OFF_GEOS_DTYPE in the Directory Header. This byte is normally 0, but may be set to 'P', to mark a disk as a Master Disk. GEOS Version 1.3 and beyond deskTops will not allow a Master Disk to be formatted, copied over, or have files deleted from the deskTop notePad. Files may still be moved to the border and deleted from there. This saves GEOS developers from having to replace application disks that have been formatted, or otherwise destroyed by user accident.

Here is the format of the BAM and Directory Header.

BAM Format/Directory Header			
	Byte	Contents	Definition
	0	18	Track of first Directory Block. Always 18.
	1	1	Sector of first Directory Block. Always 1.
	2	65	ASCII char A indicating 1541 disk format.
	3		Ignored
BAM Track 1	4		Number of sectors in Track 1
	5		Track 1 BAM sectors 0-7
	6		Track 1, BAM for sectors 8-16
	7		Track 1, BAM for sectors 17-20
BAM Track 2	4		Number of sectors in Track 1
	5		Track 1 BAM sectors 0-7
	6		Track 1, BAM for sectors 8-16
	7		Track 1, BAM for sectors 17-20
...BAM for tracks 3-34...			
BAM Track 35	4		Number of sectors in Track 1
	5		Track 1 BAM sectors 0-7
	6		Track 1, BAM for sectors 8-16
	7		Track 1, BAM for sectors 17-20
D I R E C T O R Y	144-159		Disk name, padded with CHR \$A0
	160-161	\$A0	Shifted spaces, CHR \$A0
	162-163		Disk ID word
	154	\$A0	Shifted spaces, CHR \$A0
	165-166	\$32,\$41	ASCII of \$2A, DOS version, & Disk format
	167-170	\$A0	Shifted spaces, CHR \$A0
	171-172		Tr/Sr of off page Directory Block
	173-188		GEOS ID string. "GEOS format V1.2"
	-189	0,'P'	'P' indicates protected MasterDisk
	190-255	0	Unused

The format of the Directory Block is shown below. The overall structure of a Directory Block is unchanged. The following table was taken from the c64 disk drive manual.

Directory Block Structure	
Dir. Blocks may appear on Track 18, Sectors 1 - 19	
0 - 1	Track and Sector of next Directory Block
2 - 31	Directory Entry 1
32 - 33	Unused
34 - 63	Directory Entry 2
32 - 33	Unused
66 - 95	Directory Entry 3
32 - 33	Unused
98 - 127	Directory Entry 4
32 - 33	Unused
130 - 159	Directory Entry 5
32 - 33	Unused
162 - 191	Directory Entry 6
32 - 33	Unused
194 - 223	Directory Entry 7
32 - 33	Unused
226 - 255	Directory Entry 8

Several unused bytes in each Directory Entry have been taken for use by GEOS. Bytes 1 and 2 point to the first data block in the file unless the file is a GEOS VLIR file. In this case these bytes point to the VLIR file's index table. Bytes 19 and 20 point to a new GEOS table, the File Header block as described below. Bytes 21 and 22 are used to convey the GEOS structure and type of the file. The structure byte indicates how the data is organized on disk: 0 for SEQUENTIAL, or 1 for VLIR. The file type refers to what the file is used for, DATA, BASIC, APPLICATION and other types as listed in the table below. The SYSTEM_BOOT file type should only be used by GEOS Boot and Kernal files themselves.

The TEMPORARY file type is for swap files. All files of type TEMPORARY are automatically deleted from any disk opened by the diskTop. The deskTop assumes they were left there by accident, usually when an application crashes and a swap file is left behind. When creating additional swap files, use the TEMPORARY file type and start the filename with the character PLAINTEXT. Example:

```
swapName: .byte PLAINTEXT,"My swap file",0
```

This will cause the file to print in plain text on the desk top and will prevent a user file with the same name to be accidentally removed when "My swap file" is created. Finally bytes 23 through 27 are used to hold a time and day stamp so that files may be dated.

	DIRECTORY ENTRY (Also Known as File Entry)
0	c64 File Type: 0=DELETED, 1=SEQUENTIAL, 2=PROGRAM, 3=USER, 4=RELATIVE. Bit 6=Used as Write-Protect Bit.
1	Track and Sector of First Data Block in This File.
2	If the File Is VLIR then this Word Points to the Index Table Block.
3 ... 18	16 Character File Name Padded with Shift Spaces \$A0
19 - 20	Track and Sector of GEOS File Header (New Structure)
21	GEOS File Structure Type: 0=SEQUENTIAL, 1=VLIR
22	GEOS File Types: 0=NOT_GEOS, 1=BASIC, 2=ASSEMBLY, 3=DATA, 4=SYSTEM, 5=DESK_ACC, 6=APPLICATION, 7=APPL_DATA, 8=FONT, 9=PRINTER, 10=INPUT_DEVICE, 11=DISK_DEVICE, 12=SYSTEM_BOOT, 13=TEMPORARY (for Swap Files)
23	Date: Year Last Modified, Offset from 1900
24	Date: Month Last Modified (1-12)
25	Date: Day Last Modified (1-31)
26	Date: Hour Last Modified (0-23)
27	Date: Minute Last Modified (0-59)
28 - 29	Low Byte, High Byte: Number of Blocks (Sectors) in the File

Header Block

The GEOS File Header Block was created to hold the icon picture and other information that is handy for GEOS to have around. Something worth bringing attention to is that the File Header Block is pointed to by bytes 19 and 20 of the file's Directory Entry. Thus any c64 SEQUENTIAL file may have a header block. (Bytes 19 and 20 were previously used to point to the first side sector in a c64 DOS relative file, so these bytes are unused in a SEQUENTIAL file.) Bytes 0 and 1 in any block usually point to the next block in the file, or the offset to the last data byte in the last block. The Header Block is not a file, just an extra block associated with a file. Bytes 0 and 1 are set to 00,FF, to indicate that no blocks follow.

We follow the header block diagram below by a complete description of its contents.

GEOS File Header Structure

(128 bytes. New GEOS file. Pointed to by Directory Entry)

Byte No	Contents	Description
0-1	00,FF	00 indicates this is the last block in the file. FF is the index to the last valid data byte in the block.
2	3	Width of icon in bytes, always 3
3	21	Height of file icon in lines, always 21.
4	\$80 + 63	Bit Map data type. Top bit = 1 means the lower 7 bits contain the number of unique bytes which follow, i.e. 63. Always this value.
5 - 67	\$FF,\$FF,\$FF ... \$FF,\$FF,\$FF	Start of picture data total of 63 bytes used to define icon graphic End of picture data
68	\$80 + PRG	C-64 file type, used when saving the file under GEOS PRG = 1, SEQ = 2, USR = 3, REL = 4. Bit 6 = 1 Write Protected.
69	\$02	GEOS file type: BASIC = 1, ASSEMBLY = 2, DATA = 3, SYSTEM = 4, DESK_ACC = 5, APPLICATION = 6, APPL_DATA = 7, FONT = 8, PRINTER = 9, INPUT_DEVICE = 10, DISK_DEVICE = 11, SYSTEM_BOOT=12, TEMPORARY=13
70	0	GEOS structure type, 1 = VLIR, 0 = SEQUENTIAL
71 - 72	FileStart	Start address in memory for loading the program
73 - 74	FileEnd	End address in memory for loading the program
75 - 76	InitProg	Address of initialization routine to call after loading the program
77 - 96	Filename 0,1,..,2,0,0,	20 byte ASCII application filename. Bytes 0-11 = the name padded with spaces; 12-15=version string "V1.3"; 16-20=0's
97 - 116	Parent Disk Author Name	If Data file, 20 byte ASCII filename of parent application's disk. If application program, holds name of software designer.
117 - 136	Parent Application	If Data file, 20 byte parent application filename. Bytes 0-11=name padded with spaces; 12-15=version string "V1.3"; 16-20=0's
137 - 159	Application	23 bytes for application use..
160 -255	Get Info	Used for the file menu option getInfo. String must be null terminated and null must be first char if string is empty.

Bytes 2 and 3 contain the width and height of the icon data that follows. File icons are always 3 bytes wide by 21 scan lines high. The two dimension bytes precede the data because the internal routine used by GEOS to draw icons is a general routine for drawing any size icon and it expects the two bytes to be there. Bytes 4 through 67 contain the picture data for the icon in compacted bit-map format. Byte 4 is the bitmap format byte. There are three compacted bit-map formats. The second format as described in BitmapUp, is a straight uncompact bit-map. To indicate this format, the format byte should be within the range 128 to 220. The number of bytes in the bit-map is the value of this format byte minus 128. Since the value of the highest bit is 128, the lower 7 bits, up to a value of 92 indicate the number of bytes that follow.

The lowest 3 bits of byte 68 is the old c64 file type, PRG, SEQ, USR, or REL. Byte 69 is the GEOS file type. Presently there are 11 different GEOS file types. There may be additional file types added later, but these will most likely be application data files and will be lumped together under APPL_DATA. Byte 70 is the GEOS file structure type. This is either VLIR or SEQUENTIAL. (Remember, a SEQUENTIAL GEOS file is just a linked chain of disk blocks. It does not mean a c64 SEQ file.)

Bytes 71-72 is the starting address at which to load the file. Normally, GEOS will load a file starting at the address specified in bytes 71-72. Later we will see how an alternate address can be specified. This is sometimes useful for loading a data file into different places in memory. Bytes 73-74 contain the word length address of the last byte to be loaded from or saved to disk. This word serves several purposes. First, GEOS will compute the length of the file and determine if there is enough room to save it to disk. The end-of-file address variable should, therefore, should be updated by the application in the case of a data file that grows. Second, if the file is an ASSEMBLY or BASIC file, then GEOS uses the end-of-file address to determine if it will fit in memory without wiping out the GEOS diskTurbo code. If the file fits then it can be fast loaded, otherwise GEOS will use the normal slow c64 Kernal/BASIC routines.

If the file is a BASIC, ASSEMBLY, APPLICATION, or DESK_ACC, then it is as executable file. The deskTop will look at the word comprising bytes 75-76 for the address to start execution at after the file has been loaded. Usually this is the same as the start address for loading the file, but need not be.

The next 20 bytes store the Permanent Name String. Though there are 20 bytes allocated for this string, the last 4 bytes should always be null (0). This was done to maintain compatibility with other places filenames are stored, namely in the Directory Entry.

Bytes 0-11 are used for the file name and padded with spaces if necessary. Bytes 10 to 15 should be the version number of the file. We have developed the convention that version numbers follow the format: V1.0 where 'V' is just a capital ascii V followed by the major and minor version numbers separated by an ASCII period.

Some kind of permanent name for a file is necessary since the user can rename files at will. geoWrite needs to be able to tell, for example, that a geoWrite 1.0 data file is in fact a geoWrite data file, and that it is version 1.0, even if it is named "Suzy Wong at the Beach".

Following the Permanent Name String are two strings that can be used, in the case of a data file to get to the application used to create it. Like the Permanent Name String, the first 12 characters of each of these two strings store the name and the next four characters store a version number. The last four characters are not used. The first of the 2 strings, the Parent Application Disk name in bytes 97-112, contain as you might guess, the name of disk that contains the parent application. Presently this string is not used by GEOS applications.

When GEOS needs to locate an application it looks at the the Parent Application String in bytes 117-132 . When a user double clicks on a data file, GEOS will look at the Parent Application String and try to find a file of that name. If it cannot such a file on the current disk, it will ask the user to insert a disk containing an application file of that name, "Please insert a disk with geoWrite." When looking for an application, GEOS will only check the first 12 letters of the name, the filename, and will ignore the Version Number for the time being. GEOS assumes that the user will have inserted the version of the application he wants to use. In making this assumption, GEOS tacitly assumes that applications will be downwardly compatible with data files created by earlier versions of the same application. This need not absolutely be the case as will be seen below.

When the application is loaded and begins executing, it should look at the Permanent Name String of the data file. Normally this string will be the same as the Parent Application Name with the exception that the version numbers may be different. Thus if you double click on a geoWrite V1.2 data file and insert a disk containing geoWrite V2.0, the deskTop, which doesn't compare version numbers, will load and start executing geoWrite 2.0. geoWrite will then look at the version number in the data file's Permanent Name String and determine if a conversion of data file formats needs to take place. If there were changes between the V1.2 and 2.0 versions of the data files then the data will have to be converted.

It is much more likely for the code of a program to change – to fix bugs – than it is for the data file format to change. Data format version numbers then tend to leapfrog application numbers. For example, application X starts out with V1.0. After a month of beta test V1.1 is released. After 1 week of retail shipping a bug is found and a running production change to V1.2 is made and users with V1.1 are upgraded. Meanwhile the data file format is still V1.0; any version of the application can use it. Six months later V2.0 is released with greatly expanded capabilities and a new data format. The data Version Number should then change to V2.0, leapfrogging V1.1, and V1.2. This will indicate to V1.0 to V1.2 versions of the program that they cannot read the new format. If the user has the newer version of the program than he should be using it and not an older version.

It is up to the application in its initialization code to look at the data file's version number and determine whether or not it can handle it, and if so whether or not the data needs to be converted.

Permanent Name Example

As an example, suppose the user double clicks on a geoWrite 1.0 document. The deskTop will look for a file with the name stored in the Parent Application String. If this program is not found on the current disk the deskTop will ask the user to insert a disk containing it. The deskTop only looks at the first 12 characters and will ignore the version number. After loading geoWrite, control is passed to the application. The deskTop passes a few appropriate flags and a character string containing the name of the data file. The application, in this case geoWrite, will look at the data file's Permanent Name String and especially its Version Number and determines if it can read the file, or if it needs to convert it to the more up-to-date version.. Similarly, if an older version of an application, e.g. .geoWrite 1.0, cannot read a data file created with a newer version of the application, it needs to cancel itself and return to the deskTop or request another disk.

Constants for Accessing Table Values

Constants that are used with the file system and tables described above are included in the GEOS Constants file in the Appendix. These constants make code easier to read and support, and therefore are included here. Most of the constants are for indexing to specific elements of the file tables presented above. The constants are broken down into the following sections, GEOS File Types, Standard Commodore file types, Directory Header, Directory Entry, File Header, and Disk constants.

Disk Variables

When an application first gets called there is already some information waiting for it. Several variables maintained by the deskTop for its own use are still available to the application when it is run. Other variables are set up by the deskTop in the process of loading the application. This subsection covers all the variables an application may expect to be waiting for it when it is first run. This information set up for desk accessories is slightly different. For more details on running desk accessories see the routines `GetFile` and `LdDeskAcc` later in this chapter.

Several variables necessary to talk to the drive are available to the application. There is a variable `curDrive`, and `curDevice`. `curDrive` contains the number of the drive containing the application's disk, either 8 or 9. `curDevice` is location `$00BA` where the c64 keeps the device number of the current device. When first run, `curDevice` and `curDrive` will be the same. The ID bytes for the disk containing the application are in the drive as one might expect.

Numerous variables are set up during the process of loading an application. The first group of these have to do with how the application was selected by the user. If the user double clicked the mouse pointer on a data file, GEOS will load the application and pass it the name of the data file. The application may then know which data file to use. A bit is set in `r0L` to indicate if a data has been specified. If this is the case, `r3` will point to the filename of the data file, and `r2` will point to a string containing the name of the disk which contains the data file. An application may have also been run merely in order to print a data file. Another bit is used in `r0L` to indicate this.

`r0L -loadOpt flag`

Bit 1 (application files only)

- 0 - no data file specified
- 1 - (constant for this bit is `ST_LD_DATA`) data file was double-clicked on and this application is its parent.

Bit 6 (application files only)

- 0 - no printing
- 1 - (constant for this bit is `ST_PR_DATA`) The deskTop sets this bit is set when the user clicked on a data file and then selected print from the file menu. The application prints the file and exits.

r2 and r3 are valid only if bits 1 and/or 6 in r0L are set.

r2 - Pointer to name of disk containing data file. Points to dataDiskName, a buffer containing the name of the disk which in turn contains a data file for use with the application we are loading. The application can then process the data file as indicted by bit 6 of r0L.

r3 - Pointer to data filename string. r3 contains a pointer to a filename buffer, dataFileName that holds the filename of the data file to be used with the application.

The Directory Entry and Directory Header are also available in memory as is the File Header Block.

dirEntryBuf - Directory Entry for file

curDirHead - The Directory Header of the disk containing the file.

fileHeader - Contains the GEOS File Header Block.

There is also a table created as the file is read that contains the track and sector of each block of the file. This table is called fileTrScTab. It is one block long.

fileTrScTab - List of track/sector for file. Max file size is 127 blocks (32,258 bytes).

The first word of fileTrScTab is the track/sector of File Header block. The following bytes contain the track and sector list for the remaining blocks.

r5L - Offset from the beginning of fileTrScTab to the last track/sector entry in fileTrScTab

We now turn to discussing the actual routines used to access the disk. The next section presents an overview of how to use the disk routines, and how to use the serial bus with GEOS.

Using GEOS Disk Access Routines

The GEOS Kernal contains a multitude of disk routines. These routines span a range of uses, from general powerful routines, to specific primitive routines. Most appli-

cations use only a handful out of the collection, mostly the general high-level routines. Other applications need more exacting level of disk interaction and so an intermediate level of disk access routine is provided. These are routines used by the high level routines to do what they do, and can be used to create other functions.

Finally the most primitive routines are interesting only to those who want to access a serial device other than a printer or disk drive, use the c64 DOS disk routines, or create a highly custom disk routines, a nonverified write for example.

Basic Disk Access

When running GEOS, only one device at a time may be selected on the serial bus. Usually this is one of the disk drives, A or B, but it may also be a printer or other device. The routine SetDevice is used to to change the currently selected device. You pass SetDevice the number of the device, 8 or 9 for you want to have access to the serial bus.

After selecting the device with SetDevice, call OpenDisk to initiate access to the disk. OpenDisk initializes both the drive's memory and various GEOS Kernal variables for accessing files on the disk.

Once the disk has been opened, the programmer may call any of the following high-level routines.

High-Level Disk Routines

- | | | |
|---------------|---|--|
| GetPtrCurDkNm | - | returns a pointer to a buffer containing the name of the disk. |
| SetGEOS Disk | - | Converts a normal c64 DOS disk to a GEOS disk by allocating an Off-Page Directory Block. |
| CheckDkGEOS | - | Checks to see if the current disk is a GEOS disk. |
| FindFTypes | - | Generates a list of all files on the disk of a specific type. |
| GetFile | - | Given the name of file, it will load a GEOS data file application or desk accessory files will be loaded and executed. |
| FindFile | - | Searches the disk for the file. Returns its Directory Entry. |
| DeleteFile | - | Deletes a file from the disk. |
| SaveFile | - | Saves a GEOS file to disk. |
| RenameFile | - | Gives a file a new name. |

- CalcBlocksFree - See how many free block there are left on disk.
- EnterDeskTop - Quit the application and return to the GEOS deskTop

- VLIR Routines - See the VLIR chapter.

Intermediate Routines

The routines above handle many of the functions required of an operating system, but by themselves are by no means complete. These high-level routines are implemented on top of a functionally complete set of intermediate-level routines that may be used to implement any other function needed. For example, there are no routines for formatting disks, copying disks, or copying files in the GEOS Kernal. Most applications have little need for copying disks or files and so these function were not included in the Kernal. Instead, these functions are provided by the deskTop. The deskTop is an application like any other such as geoWrite or geoPaint, except that the deskTop is a file manipulation application, and not an editor. The copy and validate functions available in the deskTop are implemented by using more the intermediate GEOS Kernal routines.

Care must be taken when using these routines to make sure that all entry requirements are met before calling them. Calling one of these routines without the proper variables and/or tables set up may trash the disk, crash the system, or both. In particular, a block is set aside in the GEOS Kernal to contain a copy of the disk's Directory Header. Some of the routines expect this block, curDirHead, to be valid, and if any values were changed by the routine it will be necessary to write the header back to disk afterwards. Below is a list in decreasing order of usefulness of these more primitive routines.

- Findfile - Returns a file's Directory Entry.
- GetBlock - Reads a block from disk
- PutBlock - Write a block to disk, and verifies it.

- GetFHdrInfo - Given a Directory Entry, fetches the file's File Header block.
- ReadFile - Reads a track/sector linked chain of blocks from disk.
- WriteFile - writes memory data out to a linked chain of blocks on disk.
- ReadByte - Simulates reading a byte at a time from a chain of blocks.

- GetDirHead - Read the Directory Header and BAM from disk.
- PutDirHead - Writes the Directory Header and BAM back to disk.

- NewDisk - Initialize the drive for reading off a new disk without changing GEOS variables.
- LdApplic - Loads and runs a GEOS application.
- LdDeskAcc - Loads and runs a GEOS desk accessory.
- LdFile - Loads a GEOS file.
- GetFreeDirBlk - Get a free Directory Entry. Allocate a new Directory Block if necessary.
- BlkAlloc,
NextBlkAlloc - Allocate a chain of blocks on the disk.
- SetNextFree - Allocates a free block on disk.
- FreeBlock - Frees up one block on disk.
- SetGDirEntry,
BldGDirEntry - Create a Directory Entry from a Header Block.
- FollowChain - Create the track/sector list in fileTrScTab for a chain of blocks.
- FastDelFile - Frees blocks indicated by the track/sector list in fileTrScTab.
- FindBAMBit - Returns information about a block.
- FreeFile - Free all blocks in a file. Leave the Directory Entry intact.
- ChangeDiskDevice - Changes the device number (8 or 9) the drive responds to.

The Most Primitive Level

An even more primitive level of routines is also available. There are only three reasons one might have for using these routines.

1. To access the standard c64 DOS routines. As mentioned before, the deskTop does this to access the formatting routines.
2. To talk to a device other than the disk drive or printer.
3. To write highly optimized disk routines for moving large numbers of blocks around that are ordered on the disk in some unusual way. The routines in the previous sections for reading and writing a linked chain of blocks on disk are almost always sufficient.

These are all ways you might want to use the serial bus that are outside the realm of what GEOS supports directly. The low level routines below are provided to allow safe access to

the serial bus, and a safe return to GEOS disk usage.

- | | | |
|---------------|---|---|
| InitForIO | - | Turn off all interrupts, disable sprites, bank switch the c64 Kernal and I/O space in. |
| DoneWithIO | - | Restore interrupts, enable sprites, and switch in the previous RAM configuration. |
| PurgeTurbo | - | Normally the turbo code is always running. PurgeTurbo removes the turbo code resident in the disk drive and returns control of the serial bus to the c64 DOS. |
| EnterTurbo | - | Uploads the turbo code to the drive and starts it running. |
| ReadBlock | - | Read a block from disk. Turbo code must already be running, and InitForIO must have been called. |
| WriteBlock | - | Write a block to disk. No verify is done, the Turbo code must be running, and InitForIO must have been called. |
| VerWriteBlock | - | Same as WriteBlock except that the block is verified after writing. |

Accessing the Serial Bus

Follow the procedure below to use the c64 serial bus.

1. Call SetDevice to set up the device you want to use. SetDevice will give the serial bus to whatever device you request.
2. If you want to use c64 DOS disk routines, then you will have to turn off the disk turbo code running in the drive. To do this, call PurgeTurbo. If not using the c64 DOS routines skip this step.
3. Call InitForIO to turn off interrupts, sprites and set the I/O space and c64 Kernal in.
4. Call any of the standard c64 DOS serial bus routines to access the serial device on the bus.
5. When finished with the bus, call DoneWithIO. This sets the system configuration back to what it was before you called InitForIO. The next GEOS disk routine that you call (except for ReadBlock, WriteBlock, or VerWrBlock) will automatically restart the diskTurbo.



High-Level File Routines

There are many routines in the GEOS Kernal for accessing and manipulating files. They range from general high-level routines to the specific primitive routines on which the others are based. Most applications will need only a handful of these routines. This chapter covers these high-level routines:

- SetDevice
- OpenDisk
- GetPtrCurDkNm
- SetGEOSDisk
- CheckDkGEOS
- FindFTypes
- GetFile
- FindFile
- DeleteFile
- SaveFile

RenameFile
EnterDeskTop
CalcBlocksFree

SetDevice

Function: Gives the serial bus to the named device

Pass: a - device number (8 - 11 for disk drives, 4 for printer)
curDevice - should already be set for the current device

Return: curDevice - new device # as was passed in a
curDrive - If the new device is a disk drive (8 - 11) then its device number is stored here
x - disk error code (see Error Codes under Constants in the Appendix)

Destroyed: a, x, y

Synopsis: Set device must be used when changing devices on the serial bus. If the present device on the bus is a disk then the turbo software running on that drive is told to relinquish the bus. The turbo code itself remains in the old drive's RAM but the drive is set back running on the normal Commodore DOS. If you want to give the bus to the printer, pass 4 in a as the device number.

OpenDisk

Function: Initialize drive for read/write of new disk. Set Device should already have been called.

Calls: NewDisk, GetDirHead, ChkDkGEOS, GetPtrCurDkNm

Pass: curDrive - should already be set to desired drive by SetDevice

Return: Dr?CurDkNm - read name of disk (as printed under icon) into this array. ? stands for A or B indicating drive A or B. Retrieved via call to GetPtrCurDkNm

r5 - points to Dr?CurDkNm. Retrieved via call to GetPtrCurDkNm
curDirHead - disk's Directory Header, retrieved via call to GetDirHead
isGEOS - if the disk is a GEOS disk this flag is set. Retrieved via call to ChkDkGEOS

x - disk error (0 if OK, see disk error values in the Appendix)
r5 - points to curDirHead where Directory Header is stored. Retrieved via call to GetDirHead

Destroyed: a, x, y, r0 - r4, r6 - r15

Synopsis: OpenDisk is used to initiate access to the disk in the current drive. OpenDisk is meant to be called after a new disk has been inserted in the disk drive, and after SetDevice has been called to give the bus to the correct drive. OpenDisk calls NewDisk to initialize the disk drive for the new disk. The ID byte for the disk is stored in the drive. Next it calls GetDirHead to read in the Directory Header into curDirHead. It then calls GetPtrCurDkNm to load the disk's name (the same character string that appears under the disk icon) into the Dr?CurDkNm array. (The ? stands for A or B depending on which is the current drive.) R5 is left pointing to Dr?CurDkNm. OpenDisk calls ChkDkGEOS and sets the isGEOS flag if this is a GEOS disk. The turbo code is installed in the drive, if it is not already there, and left running for future accesses to the disk.

GetPtrCurDkNm—Get Pointer to Current Disk Name

Function: Return a pointer to either DrACurDkNm or DrBCurDkNm: the name of the disk in the current disk drive.

Called By: OpenDisk

Calls: BBMult

Pass: x - address of zpage register in which to return pointer diskName string, i.e. , ldx #r8 will cause GetPtrCurDkNm to use r8. Note: r15 may not be specified

curDrive - number of the **current disk drive**, usually 8 or 9

DrACurDkNm - the disk name for drive A

DrBCurDkNm - the disk name for drive B

Return: pointer to Dr?CurDkNm in the **register** specified by x register

Destroyed: a, y

Synopsis: GetPtrCurDkNm (Get Pointer to Current Disk Name) uses curDrive to determine which is the current disk drive, A or B. It then returns the address of DrACurDkNm or DrBCurDkNm in the register whose address was passed in x. To use this routine just pass in x the address of the register, ldx #r8 for example, to get the pointer back in r8. Any register except R15, which is used internally may be specified. The names for the current disks should already be stored in DrACurDkNm or DrBCurDkNm. The specified register is returned with a pointer to either DrACrDkNm or DrBCurDkNm.

SetGEOSDisk

Function: Writes the GEOS ID string into the Disk's Directory Header, creates off-page Directory Block, writes Directory Header and BAM back to disk.

Calls: GetDirHead, CalcBlksFree, SetNextFree, PutDirHead

Pass: curDrive - number (8 or 9) of current drive being used

Return: curDirHead - Contains GEOS ID String and a track/sector pointer to the off-page Directory Block
disk - Directory Header updated, off-page directory block added.

Destroyed: a, x, y

Synopsis: The purpose of SetGEOSDisk is to convert a c64 disk to a GEOS disk by adding an ID string and off page directory block. SetGEOSDisk calls GetDirHead to read the Directory Header for the current disk into curDirHead. It writes the GEOS ID string to an unused portion of the Directory Header. It also assigns a sector on the disk (via CalcBlksFree and SetNextFree) to contain the OFF-PAGE Directory Block, and stores a pointer to it in the curDirHead in bytes 171-172. SetDevice must have been executed sometime before the call to this routine so that curDevice has the correct drive number, and the drive has the serial bus.

ChkDkGEOS

Function: Checks for GEOS ID string in current disk to see if disk is a GEOS created disk.

Called By: OpenDisk

Pass: r5 - pointer to **Directory Header** image in memory, usually at curDirHead
Directory Header - expected in memory, usually at curDirHead

Return: isGEOS - TRUE if **GEOS created** disk
a - same as isGEOS

Destroyed: x, y

Synopsis: ChkDkGEOS (Check Disk GEOS) checks if the disk in the current drive is a GEOS created disk by checking the GEOS format string in the Directory Header. If the string is found, the flag isGEOS is set to TRUE. Use GetDirHead to read in the Directory Header.

FindFTypes

Function: Returns the names of files of a given file type.

Used By: DBGETFILES dialog box routine

Pass:

- r6 - pointer to filename buffer; 16 characters max + 0 as the 17th
- r7L - GEOS file type to search for
- r7H - maximum number of filenames to list
- r10 - 0 for ignore permanent name or a pointer to null-terminated string, maximum 16 characters, to match against permanent name string in File Header blocks of the files FindFTypes will be checking on disk.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Return: filename buffer gets filenames of matched files

Destroyed: a, y, r0, r1, r2L, r4, r6

Synopsis: FindFTypes builds a list of files that match the requested GEOS file type. The caller passes the file type, a pointer to a buffer in which to place the null-terminated filenames, and the maximum number of files to list. Each filename is padded with zeros out to 17 characters. The 17th character is always the null terminator, 0. The filenames are checked in the order they appear on the disk, which is also the order they appear on the deskTop notepad. Current file types are: NOT_GEOS, BASIC, ASSEMBLY, DATA, and the GEOS supported file types, SYSTEM, DESK_ACC, APPL_DATA, FONT, PRINTER, INPUT_DEVICE, DISK_DEVICE, and TEMPORARY. See the section on file types or the Constants listing for an explanation of each type.

For example, when an application is loaded, FindFTypes is used to build the list of desk accessories available to the user (only the first eight

desk accessories on the disk at the time of boot up are available). FindFTypes is passed the DESK_ACC file type, the number of files to get, MAX_DESK_ACC=8, and a pointer to a filename buffer.

The entire filename buffer is cleared before getting the filenames. This way a zero as the first character in a filename indicates the end of the list. Furthermore r7H is decremented each time a file type match occurs, and can be used to compute the number of files found: maxnames - r7H.

A further option is available with GEOS supported file types. It allows files of a specific GEOS type, for example, APPLICATION_DATA, to be screened so that only those files created by a particular application be selected. FindFTypes does this by checking the **permanent name string** as stored in the file's FileHeader block. An application created by geoPaint, for example, will have the permanent name **geoPaint V1.2**. There are extra spaces after geoPaint in order to pad the V1.2 out to be the 13-16 characters in the string. (Even though there are 20 bytes in the File Header block for the permanent name string, only 16 are used.) The caller passes a pointer in r10 to a null-terminated string to match against the permanent names of files with the proper GEOS file type. Only the number of characters in the string pointed to by r10 are checked. This is so that the suffix of a permanent name, typically a version number like Vn.n, can be ignored if desired.

GetFile

Function: The master load and run routine.

Calls: FindFile, and one of LdFile, LdDeskAcc, or LdApplic

Pass: r6 - pointer to filename string. The string must be null terminated and no longer than 16 chars, excluding the terminator.

r0L - **loadOpt** flag

Bit 0

0 - follow standard loading for file

1 - (constant for this bit is ST_LD_AT_ADDR) load file at address specified in loadAddr

Bit 1 (application files only)

0 - no data file specified

1 - (constant for this bit is ST_LD_DATA) data file was double-clicked on and this application is its parent.

Bit 6 (application files only)

0 - no printing

1 - (constant for this bit is ST_PR_DATA) The deskTop sets this bit is set when the user clicked on a data file and then selected print from the file menu. The application prints the file and exits.

r2 and r3 are passed only if:

1. an application is to be loaded and
2. bits 1 and/or 6 in r0L are set:

r2 - Pointer to name of disk containing data file. Points to **data DiskName**, a buffer containing the name of the disk which in turn contains a data file for use with the application we are loading. r2 and the contents of dataDiskName are forwarded to the application. The application can then process the data file as indicted by bit 6 or 7 of LoadOpt.

r3 - Pointer to data filename string. r3 contains a pointer to a filename buffer, **dataFileName** that holds the filename of the data file to be used with the application. r3, and the contents of

dataFileName are forwarded to the application. The application can then process the data file as indicted by bit 6 or 7 of LoadOpt.

r7 - Contains the **load address** when the ST_LD_AT_ADDR (load at address option) bit is set in r0L.

r10L - **DA Recover flag**. When the file to be loaded is a desk accessory, r10L contains a flag, which is forwarded to it.

Bit 7 foreground bit

0 - doesn't save foreground screen data.

1 - (constant for this bit is FG_SAVE) save the foreground screen data, upon entry to DA and restore upon exit.

Bit 6 color bit

0 - doesn't save color

1 - (constant for this bit is CLR_SAVE) save the color information upon entry and restore upon exit.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Return: Returns the file in memory at location indicated in the file's Header Block or the alternate address passed in r7. If the file was an application or desk accessory, it is run; the system is reinitialized and execution begins at the start address as indicated in the File Header. In the case of a DA or application, GetFile never returns to its caller.

Applications : r2, r3, r7 and r0L are passed on to the application or DA. In the case of an application, r7 is contains the initialization vector (start address) as taken from the fileHeader in stead of the the load address.

dataDiskName, loadAddr, and dataFileName are unchanged.

x - error code, if disk error, or file not found. This error will force GetFile to return to caller (obviously). See disk errors in the Appendix.

From call to **FindFile**: **dirEntryBuf** - Directory Entry for file

From call to **GetFHdrInfo** via **LdFile**, **LdApplic**, or **LdDeskAcc** (**LdApplic** in turn calls **LdFile**): **fileHeader** - Contains the GEOS File Header Block, even if file is a VLIR file.

From **ReadFile** via **LdFile**, **LdApplic**, or **LdDeskAcc**:

fileTrScTab - List of track/sector for file or record. Max file size is 127 blocks (32,258 bytes). **GetFileHdr** fills in the first word of **fileTrScTab** with the track/sector of File Header block. Thus when **GetFile** is used to load a file, **ReadFile** is called to complete the **fileTrScTab**. If the file is VLIR, then bytes 2, 3 and the following bytes contain the track and sector list for the first **record** in the file.

r1 - in case of a disk BUFFER-OVERFLOW error only, **r1** contains the track/sector of the block which didn't fit and was not read in. Not returned if called via **LdDeskAcc**.

r5L - offset from the beginning of **fileTrScTab** to the last (word length) entry in **fileTrScTab**

From call to **LdDeskAcc**: **font** - Font set to system font for entry to DA

LdApplic and **LdDeskAcc** do a **warm start** initialization of the system. See the **System Warm-Start Configuration** in Chapter 19 for the state of system variables.

Destroyed: **a**, **y**, **r0** - **r10**. Buffers File Header, **dirEntryBuf**, **curDirHead**, are not affected.

Synopsis: **GetFile** requires only a filename to load any GEOS file type. If the file is a data file it is loaded into memory at the location specified in its Header Block. (See the discussion on file structure.) If the file is an application, it is loaded at the address specified by the File Header Block, and run.

The disk is searched for the file and, if found, it is loaded with the proper routine: for example, a desk accessory requires a different loading procedure than a data file. If the disk is a GEOS disk, the off page directory file is also searched to locate the file.

If a file is a data file, but what is desired is that the application for that data file be run, then the proper bits in the **loadOpt** flag may be set to do

that. As an example, the deskTop uses GetFile to load applications when one of the following happens:

1. the application is double clicked,
2. a data file created with the application is double clicked,
3. a data file is selected and print is chosen from the file menu.

If the deskTop is loading an application because a data file was selected, or if a data file was selected for printing via the file menu on the deskTop, then

1. r2 should point to a string in memory containing the name of the disk which contains the data file.
2. r3 should point to a string containing the filename of the data file.

This is necessary in case the deskTop had to open another disk in order to find the application, and in order to pass the data filename on to the application so that it can load the data file itself. If the loadOpt flags don't indicate either of these cases, then GetFile and any application it loads should ignore r2 and r3.

If a Desk Accessory is being requested then r10L should specify whether or not the DA is required to do a RecoverRectangle to recover the background screen to the foreground screen and therefore restore the appearance of the calling application. Some applications do not use the background screen to buffer the graphics on the foreground screen and therefore the DA should leave well enough alone. In these cases the applications will restore the foreground screen themselves.

FindFile

Function: Loads the Directory Entry for a file with the indicated filename.

Called by: GetFile, FindFile, FastDelFile, RenameFile

Pass: r6 - pointer to string containing the **filename**. The string must be null terminated and no longer than 16 chars, excluding the terminator.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Accessed: disk - Directory Entries on disk

Return: x - error code, if disk error, or file not found. See disk error discussion.
diskBlkBuf - **Directory Block** containing Directory Entry for file.
dirEntryBuf - **Directory Entry** for file
curDirHead - The **Directory Header** of the disk containing the file.
r1 , r1H - Track and Sector of directory block on disk
r5 - pointer to directory entry within diskBlkBuf

Destroyed: a, y, r4, r6

Synopsis: FindFile requires only a 16 character filename and a currently open disk to find a file. Thus an OpenDisk or NewDisk must have been done before calling this routine. (OpenDisk and NewDisk in turn require that a SetDevice must have been done.) The disk is searched for a file with the given filename. If the file is found, its Directory Block and Directory Entry are loaded for use with other reading and writing routines. Check x for file not being found or other disk errors. The filename to search for is typically one of a number of filenames returned by FindFTypes or a filename typed in by the user.

SaveFile

Function: Save a region of memory onto disk as a GEOS SEQUENTIAL file. SaveFile is also the routine to call to create an empty VLIR file.

Called By: Any routine needing to create a GEOS file.

Calls: GetDirHead, SetGDir Entry, PutDirHead

Pass: r9 - pointer to **File Header** block. The first two bytes of the file header when stored to disk will contain (00, FF). When passed to SaveFile though, these two bytes should point to a null terminated string containing the filename. See the File Structure chapter for more details on the File Header Block.

r10L - number of the **directory page** on deskTop Notepad to try and put the file on. One directory page is stored in each directory block on track 18 on the disk. For example if you pass 4 in r10L, SaveFile will try to put the file on page 4 on the deskTop.

fileTrScTab - The **track and sector list** of all the blocks in the file.

Return: fileHeader - Contains **File Header** as written to disk.
 curDirHead - current **Directory Header** via call to GetDirHead
 r6 - pointer to **fileTrScTab**
 r9 - pointer to **File Header** block
 fileTrScTab - unchanged
 dirEntryBuf - has new **Directory Entry** for the file
 DirectoryHeader - New **Directory Header and BAM** written out to disk.
 New File Header and file written out. Index Table for VLIR file written.
 Index Table - If file is VLIR then an Index Table is created for it.
 x - disk status nonzero if error.

Destroyed: a, y, r0 - r8

Synopsis: SaveFile will save a region of memory onto disk as a GEOS SEQUENTIAL file. To do this it needs a GEOS File Header and a page number on the deskTop notepad. The GEOS File Header contains most of the information SaveFile needs to create a Directory Entry for the file and save it to disk: the icon, the file types, the load and run addresses, the version string. See a description of File Headers in this manual.

The only piece of information which does not normally appear in the File Header but which is needed in the Directory Entry is the Filename.

SaveFile, therefore, expects the first two bytes of the File Header, which stored on disk to instead point to a null terminated string to be used for the filename. Filenames may be up to 16 characters long, followed by a 0 as terminator.

SaveFile can be used to create an empty a VLIR file. The only difference in this case is that the start address for the file should contain 0 and the end address \$FFFF. This is so that no data blocks actually get allocated to the file. The GEOS file type in the FileHeader passed to **SAveFile** must indicate that the file is VLIR. When **SaveFile** calls **SetGDirEntry**, a block for the Index Table will get allocated. **SaveFile** then writes an empty Index Table to that block.

Each page on the deskTop notepad corresponds to one Directory Block in Track 18 of the disk. The number passed in **r10L** is the page number in the notepad to try and put the file. (The third page in the notepad is *not* stored in the third Directory Block on the disk due to disk storage constraints (interleave).) If the requested page is full, **SaveFile** will keep looking on later pages until it finds an available Directory Entry to use for the file. If it has to create a Directory Block, and page in the notepad, it will.

DeleteFile

Function: Delete a c64, GEOS SEQUENTIAL, or VLIR file.

Called By:

Calls: FindFile, FreeFile

Pass: r0 - pointer to the filename, (null terminated)

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primative routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Return: x - Error status, 0 = OK, see Appendix for disk errors
turbo - Turbo code turned off but not purged from drive
Directory Header/BAM - **curDirHead** is updated to indicate the newly freed blocks and written out to the Directory Header/BAM block on disk.
Directory Block - The Directory Block containing the Directory Entry for the deleted file is written back out to disk.
File Header Block/VLIR Index Table - Removed from disk along with rest of file.
dirEntryBuf - Returned from call to FindFile for the deleted file.
File Header Block/VLIR Index Table - Removed from disk along with rest of file.
r9 - Points to dirEntryBuf

Destroyed: a, y, r0 - r9

Synopsis: Delete a file with the given filename from the current Disk. Disk Turbo routines are used to delete the sectors comprising the file. No Commodore DOS routines are used. Any size or type of file including VLIR files may be deleted with DeleteFile. The Directory Entry on disk is removed but the copy of it in dirEntryBuf remains. Each sector in the file is

marked in the BAM as being free. If the Relative file side-sector pointer is nonzero, then it either points to a side sector chain in a relative file or to a File Header if it's a GEOS file. In either case whatever it points to is freed up. VLIR files have their index table, FileHeader, and all records deleted.

RenameFile

Function: Renames a file with a new name.

Calls: FindFile

Pass: r6 - pointer to **old filename** (null terminated)
r0 - pointer to **new filename** (null terminated)

Return: **dirEntryBuf** - Contains the directory entry for the file with the new filename.

diskBlkBuf - Holds the Directory Block containing the file's Directory Entry. The file name string is changed and diskBlkBuf is written back out to disk.

x - error status, 0 =OK; see Disk Errors in the Appendix on Constants.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Destroyed: a, x, y, r1, r4 - r6

Synopsis: Given nothing but the new and old filenames for a file, rename the file. FindFile gets the Directory Entry for the file with the given name. It will then replace the filename in the Directory Entry with the provided new file name and write it back. The filename pointer passed in r6 must not point within dskBlkBuf as it will get trashed when FindFile reads in the Directory Block.

EnterDeskTop

- Function:** Reinitializes the GEOS system and begins execution of the deskTop.
- Called By:** Applications upon exit
- Pass:** nothing
- Return:** A warm start initialization is performed. See FirstInit in this section.
- Destroyed:** System variables reinitialized.
- Synopsis:** When an application exits, it should execute a **jmp** to EnterDeskTop. This will load the deskTop program and reinitialize the GEOS system.

CalcBlksFree

Function: Calculates the number of free blocks on disk by looking at BAM.

Called By: BlkAlloc, NxtBlkAlloc, SetGEOSDisk

Pass: r5 - pointer to **Directory Header** (usually curDirHead)

Return: r4 - the **number of free blocks** on the disk
r5 - unchanged

Destroyed: a, y

Synopsis: Given the current disk's Directory Header, CalcBlksFree searches through the BAM and counts up the number of free sectors on the disk. There are 35 tracks on a disk. Within the BAM there is one BAM entry for each track. Each BAM entry is four bytes long. The first byte contains the number of free blocks on the track. The remaining three bytes contain one bit for each block on the track. The BAM takes us byte \$4 to \$8F (4 to 143) in the Directory Header, track 18, sector 0.

15.

Intermediate Level

In the previous chapter we covered the high-level GEOS file routines. This chapter describes the intermediate-level routines. These routines can be used construct functions that were impossible to fit into the Kernal such as disk and file copy. Such functions are easy to construct and may be sometimes be optimized for the current application. They include:

FindFile	GetFreeDirBlk
GetBlock	BlkAlloc
PutBlock	NxtBlkAlloc
GetFHdrInfo	SetNextFree
ReadFile	FindBAMBit
WriteFile	FreeBlock
ReadByte	SetGDirEntry
GetDirHead	BldGDirEntry
PutDirHead	FollowChain
NewDisk	FastDelFile
LdApplic	FreeFile
LdFile	

FindFile

Function: Returns a file's directory entry. (See the high-level routines section for this function.)

GetBlock

Function: Standard primitive routine for fetching a block off of disk.

Called By: GetFHdrInfo, UpdateRecordFile, update this entry

Calls: EnterTurbo, InitForIO, ReadBlock, DoneWithIO

Pass: r1L, r1H - track/sector of first block to read
r4 - pointer to buffer in which to store data from disk, usually diskBlkBuf.

Initialized Drive - The drive must have been initialized via OpenDisk, or NewDisk, and must be the selected device on the serial bus, via a call to SetDevice.

Return: x - error status, 0 = OK, see Appendix for disk errors
r4 - pointer block read in (unchanged)
r1 - unchanged

Destroyed: a, x, y, r0 - r15

Synopsis: Given the track/sector of the block to read in, GetBlock will read a block into memory at the designated buffer area. GetBlock loads a sector from the current 1541 drive into c64 memory using the Turbo software. After being read, the block **still contains the track/sector** pointer which takes up the first word in the sector on disk. GetBlock is a low level routine, and most useful reading a single block into diskBlkBuf, or supporting custom disk routines, something not especially recommended for the inexperienced.

GetBlock will ,
 turn off interrupts,
 turn off sprites,
 turn on the turbo code in the drive,
 (tranferring it there if needed),
 read the block,
 turn off the turbo code,
 turn on sprites, and
turn on interrupts.

GetBlock always transfers 256 bytes from each sector, even for the last sector of the chain. GetBlock in the V1.2 Kernal only transmitted the used bytes in the last block. This was changed in order to be able to read disks that consist of data blocks not using the standard T,S linkage, such as a few graphics disks available for the c64.

PutBlock

Function: Write memory block (which already contains track/sector info) to disk.

Calls: EnterTurbo, InitForIO, WriteBlock, DoneWithIO

Pass: r1L, r1H - track/sector of location of first sector to write to.
r4 - pointer to the data in memory, usually diskBlkBuf.

Initialized Drive - The drive must have been initialized via OpenDisk, or NewDisk, and must be the selected device on the serial bus, via a call to Set-Device.

Return: x - error status, 0 = OK, see Appendix for disk errors.
r4 - unchanged
r1 - unchanged

Destroyed: a, x, y, r0 - r15

Synopsis: PutBlock is the disk primitive that writes a block from the c64 memory to the currently open 1541 drive using the Turbo software. For a file to be stored correctly, blocks must be linked, that is, the first two bytes of each block while still stored in c64 RAM should be updated to contain the track and sector where the next block will be stored. PutBlock may then be called to write the block to disk.

The block to write is usually stored in diskBlkBuf. PutBlock is often used in a loop which gets the next free block on disk, updates the T/S pointer in the RAM block and calls PutBlock to write the block out to disk.

PutBlock is probably only used after a small alteration of a file read by GetBlock. If blocks are added to a file, then NxtBlkAlloc must be called to allocate free sectors from the BAM on disk. The track/sector bytes must then be inserted as the first word of each block. PutBlock is a low level routine and probably only useful to a programmer developing custom disk routines. Usually one of the higher level file routines is what is needed.

PutBlock, like GetBlock, will ,
 turn off interrupts,
 turn off sprites,
 turn on the turbo code in the drive,
 (tranferring it there if needed),
 read the block,
 turn off the turbo code,
 turn on sprites, and
 turn on interrupts.

PutBlock always transfers 256 bytes from each sector, even for the last sector of the chain. PutBlock in the V1.2 Kernal only transmitted the used bytes in the last block. This was changed in order to be able to read disks that consist of data blocks not using the standard T,S linkage, such as a few graphics disks available for the c64.

The turbo code in the drive will hold the serial bus so no other device may access the serial bus while disc transfers are taking place.

GetFHdrInfo

Function: Load File Header Block into fileHeader

Called By: LdFile, LdDeskAcc

Calls: GetBlock

Pass: r9 - pointer to Directory Entry for file, usually stored in dirEntryBuf.
dirEntryBuf - usual buffer for holding the Directory Entry. Even though GetFHdrInfo doesn't need the whole Directory Block, it must have been read anyway (e.g., FindFile) in order to get the Directory Entry, and is usually in memory at diskBlkBuf.

Return: r1 - track and sector of **first data block** in file copied from File Header. If VLIR file then this contains the track and sector of the Index Table block.
r7 - **start address** of the file, retrieved from File Header
fileHeader - loaded with the file's **File Header block**
fileTrScTab - the first two bytes are the **track/sector** of the File Header block
x - **disk error** flag, 0 = all OK, for disk errors see the Appendix

Destroyed: a, y, r4

Synopsis: GetFHdrInfo retrieves the track and sector of the File Header block from the Directory Entry and loads the File Header into **fileHeader**. r1 is left with the track and sector of the first data block in the file, while r7 returns the load address for the file.

ReadFile

Function: File Reading primitive for reading in a linked chain of blocks from the disk. The track/sector pointer bytes are discarded.

Called By: LdFile, LdDeskAcc

Calls: EnterTurbo, InitForIO, ReadBlock, DoneWithIO.

Pass: r7 - address in memory at which to load a linked chain of blocks.
r1L,r1H - track/sector of first block to load, either the first block of the file or in .
r2 - size of destination buffer in bytes.

Return: fileTrScTab - track/sector list for file or record. Max file size is 127 blocks (32,258 bytes). GetFHdrInfo fills in the first word of **fileTrScTab** with the track/sector of File Header block. Thus when GetFile is used to load a file, ReadFile is called to complete the fileTrScTab.

x - **error status:** if the file is too large to fit in the buffer of size indicated in r2, a BUFFER_OVERFLOW error will result. The offending block is not read. If no error then x is 0.

r7 - points in memory to the byte following the **last byte** read in.

r5L - offset to the **last (word length) entry** in fileTrScTab

r1 - in case of BUFFER_OVERFLOW, r1 contains the track/sector number of the **block which didn't fit** and was not read in.

Destroyed: a, y, r1 - r4

Synopsis: ReadFile reads a data file into memory and finishes building the track/sector table in fileTrScTab. ReadFile is usually called as a result of a call to one of the higher level file loading routines such as GetFile, LdFile, or LdApplic. GetFile calls LdFile which calls ReadFile to read both regular Commodore files and VLIR files. In the case of a VLIR file, the track and sector of the first record is passed in r1. Before its call to ReadFile, LdFile calls GetFileHdrInfo to get the fileHeader. From the fileHeader, LdFile gets the track/sector for loading the Index Table and gets from the Index Table the track/sector of the first record. In the case of a VLIR file the track/sector of the first record is passed to ReadFile in r1. ReadFile stores r1 in fileTrScTab+2,3. The first two bytes of fileTrScTab contain the track and sector of the File Header block as returned by GetFHdrInfo.

WriteFile

Function: Writes a section of memory from memory to disk using a pre-allocated Track/Sector chain of free blocks on the disk. All blocks written are verified.

Called By: SaveFile

Calls: disk Turbo code

Pass: r7 - pointer to beginning of the **data** area to write to disk
r6 - pointer to **table of free disk blocks**, (fileTrScTab)

Initialized Drive -- In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up accesses.

Return: x - **error status**, zero for no error, see Appendix for disk errors
fileTrScTab - **free block** Track/Sector allocation table unchanged

Destroyed: a, y, r1, r2, r4, r6, r7

Synopsis: WriteFile does the actual transfer of data from memory to the disk, all the hard work has already been done: the Directory Entry should have been allocated with SetGDirEntry, BldGDirEntry, and/or GetFreeDirBlk, and the T/S table of free blocks in fileTrScTab allocated with BlkAlloc. To use these routines the File Header block had to be created. WriteFile itself needs only fileTrScTab, r6, and r7.

WriteFile starts writing data from memory at the location passed in r7. Everytime it gets a block from memory it looks in fileTrScTab for the track and sector of the next allocated block on disk. 254 bytes of data are appended to the T/S of the next block in the chain, and written to disk. The track and sector of the next block allocated are retrieved from

fileTrScTab. The end of the file is determined by the last block allocated in fileTrScTab. The first two bytes of this block will not point to the TS of the next block on disk but instead have the value \$00,\$(index to last used byte in block).

ReadByte

Function: Allows a linked list of blocks on the disk to be read one byte at a time. **r1, r4, r5 and the disk buffer must not be destroyed** between calls to ReadByte.

Calls: GetBlock

Pass: For initial call:
r1 - track/sector of **first block** to read
r4 - pointer to a **one block buffer**, usually **diskBlkBuf**
r5 - 0, index to the **next byte** to read

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk.

Return: r1 - track/sector of **next block** to read
r4 - unchanged
r5 - incremented, index to the **next byte** to read
x - **disk error status**, 0 =OK, attempt to read past last byte in linked chain results in a BUFFER_OVERFLOW error.
a - the **data byte** from the file
Zero Flag - the Z flag in the status register is set for contents of x, this makes for quick check of error status. (If zero then OK.)

Destroyed: y only

Synopsis: ReadByte allows a linked list of blocks on the disk to appear to be read one byte at a time. The input is actually buffered one block at a time. Both standard files and VLIR files may be read in this manner. The caller passes ReadByte the track/sector of the first block in the chain. You may call ReadByte over and over, each time receiving one more byte of the file -- the only catch is that you must preserve the disk buffer and registers r1, r4, and r5 between calls to ReadByte.

GetDirHead

Function: Get the Directory Header and BAM for the current disk into memory.

Called By: OpenDisk, SetGEOSDisk, BlkAlloc, FreeFile, FastDelFile

Calls: GetBlock

Pass: curDrive - Contains 8 or 9, for the number of the currently active drive. The drive must be the current device on the serial bus by virtue of a call to SetDevice, and should have been initialized with a call to NewDisk (or OpenDisk which calls NewDisk).

Return: curDirHead - loaded with the current disk's **Directory Header**

Destroyed: a, x, y, r1, r4, others

Synopsis: GetDirHead reads the Directory Header, including the BAM, from the current Disk into curDirHead. A SetDevice, must have already been done on the current disk to load curDrive with the proper drive number and give the bus to that drive. A NewDisk (or OpenDisk) should have been done to initialize the disk drive for access.

This routine will be extended to read in all portions of the BAM for other drive types. GEOS keeps the BAM in memory while allocating and freeing blocks on the disk. When changed, the BAM must be written back to the disk with PutDirHead. This is the reason that the user must not remove the disk from the drive between accesses. The disk will be trashed if the new BAM is not written back out.

PutDirHead

Function: Write the current Directory Header back out to disk.

Called By: SaveFile, FreeFile, FastDeleteChain, SetGEOSDisk

Calls: setBAMParems, PutBlock

Pass: curDirHead - valid Directory Header for current disk
curDrive - Contains 8 or 9, for the number of the currently active drive.
The drive must be the current device on the serial bus by virtue of a call to SetDevice, and should have been initialized with a call to NewDisk (or OpenDisk which calls NewDisk).

Return: disk - Directory Header on disk written from curDirHead back to disk.

Destroyed: a, x, y, r0 - r15

Synopsis: Write the current Directory Header stored in curDirHead out to the current disk. PutDirHead will be extended to write bak all portions of the BAM for new drive types.

NewDisk

Function: Load the BAM from the *current* disk into the Drive's internal memory.

Called by: OpenDisk

Calls: EnterTurbo, InitForIO, DoneWithIO

Pass: curDrive - must have proper device number

Return: DriveMemory - New BAM read into the drive
x - error status, 0 = OK; see Error Codes in the Appendix on Source.

Destroyed: a, y, cmdnBuff, r1

Synopsis: The 1541 drive stores the ID byte from the inserted disk in its memory. When a new disk is inserted, a new ID byte must be read into the drive's memory before any files can be read. NewDisk tells the drive to read the BAM for the disk just inserted. The disk drive should already be listening to the bus, so NewDisk often follows SetDevice.

If the turbo code is not running in the drive, it is installed there and left running. NewDisk is called from within OpenDisk.

LdApplic - Load Application File

Function: Loads and runs a GEOS application file.

Called By: GetFile

Calls: LdFile

Pass: r9 - pointer to Directory Entry for file, usually points to dirEntryBuf
 r0L - **loadOpt** flag

Bit 0

- 0 - follow standard loading for file
- 1- (constant for this bit is ST_LD_AT_ADDR) load file at address specified in loadAddr

Bit 1 (application files only) Passed through to application

- 0 - no data file specified
- 1- (constant for this bit is ST_LD_DATA) data file was double-clicked on and this application is its parent.

Bit 6 (application files only)

- 0 - no printing
- 1 - (constant for this bit is ST_PR_DATA) this bit is set when the application is requested to print the file and exit.

r2 - Pointer to name of disk containing data file. Points to dataDiskName, a buffer containing the name of the disk which in turn contains a data file for use with the application we are loading. r2 and the contents of dataDiskName are forwarded to the application. The application can then process the data file as indicated by bit 6 or 7 of LoadOpt.

r3 - Pointer to data filename string. r3 contains a pointer to a filename buffer, dataFileName that holds the filename of the data file to be used with the application. r3 and the contents of data FileName are forwarded to the application. The application can then process the data file as indicated by bit 6 or 7 of LoadOpt.

r7 - Contains the load address when the ST_LD_AT_ADDR, (load at address option) bit is set in r0L.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable

curDrive, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Return: Returns the application file in memory, or if it is a VLIR file, the first record. Unless the load at address option in loadOpt is specified, LdApplic reinitializes the system and runs the application: i.e. it never returns. When the "load at address" option is set, LdApplic returns to the original caller as it would not make sense to start executing an application that had been loaded to some other location than the one it was saved from -- it is unlikely that the application would work.

loadAddr - the alternate load address
loadOpt= r0L - the load option flag

r7 has the **initialization vector** (start address) as taken from the file's File Header

r2, r3, and r0 are passed on to the application or DA.

r5L - offset from the beginning of fileTrScTab to the last (word length) entry in fileTrScTab

dataDiskName, **loadAddr**, **dataFileName**, and **dirEntryBuf** are unchanged.

fileTrScTab+2 - list of track/sector for file or record. Max file size is 127 blocks (32,258 bytes). GetFileHdr fills in the first word of **fileTrScTab** with the track/sector of File Header block. Thus when GetFile is used to load a file, ReadFile is called to complete the fileTrScTab. If the file is VLIR, then bytes 2,3 and the following bytes contain the track and sector list for the first **record** in the file.

x - **error status**: for example, if the file is too large to fit in this space, a BUFFER_OVERFLOW error will result. The offending block is not read. See disk errors in the appendix. A disk error will force a return to the calling routine (the routine that called LdApplic).

r1 - In case of a disk BUFFER-OVERFLOW error, r1 contains the track/sector of the block which didn't fit and was not read in.

LdApplic does a **warm start** initialization of the system. See the **System Warmstart Configuration** Appendix for the state of system variables.

Destroyed: All pseudoregisters not listed under return above. Unless "load at address" option is specified, LdApplic does not return.

Synopsis: LdApplic is called from GetFile to load and initialize a new GEOS application. GetFile calls FindFile before LdApplic. This is why LdApplic depends on the variables returned by FindFile. An application may be invoked in three ways.

1. The application is invoked directly as happens when a user double clicks its icon on the deskTop.
2. The application is invoked through a data file as happens when the user double clicks on the icon for a data file created by the application.
3. The application is invoked to print a data file: the user selects a data file and chooses print from the file menu.

The loadOpt flag is passed to LdApplic indicating how the application was invoked. If the application is invoked through a data file or for printing, the calling routine, usually the deskTop, must pass the diskname, and filename of the data file. It is up to the application to read loadOpts and read in or print the data file.

Finally, it is possible to load an application to an address other than the one it was saved form. This option is more useful for data files, for it is unlikely that the applicaton will run unless it contains entirely relocatable code. (This may become useful for application switching when RAM expansion is available for the c64.) This is the "load at address" case and since it is so unlikely that the caller actually wanted the application run at the alternate address, LdApplic returns to the caller instead of executing it. Note that it is likely that the application is likely to be large enough to trash the caller's code as it is loaded.

LdFile - Load File

Function: Loads a file, usually called from GetFile or LdApplic

Called By: GetFile, LdApplic

Calls: GetFHdrInfo, ReadFile

Pass: r9 - pointer to **Directory Entry** usually in dirEntryBuf
 loadOpt - flag for **loading option**
 Bit 0
 0 - follow load information in File Header Block
 1 - load file at address specified in loadAddr. Constant for this is ST_LD_AT_ADDR.
 loadAddr - (**optional**) if ST_LD_AT_ADDR set in loadOpt, loadAddr should contain the **loading address** for the file.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Return: The File - loaded in memory. Returns to calling routine.
 loadOpt - flag for **loading option** unchanged
 loadAddr - **alternate load address** unchanged
 fileTrScTab+2 - list of track/sector for file or record. Max file size is 127 blocks (32,258 bytes). GetFileHdr fills in the first word of **fileTrScTab** with the track/sector of File Header block. Thus when GetFile is used to load a file, ReadFile is called to complete the fileTrScTab. If the file is VLIR, then bytes 2,3 and the following bytes contain the track and sector list for the first **record** in the file.
 x - **error status**: if the file is too large to fit in this space, a BUFFER_OVERFLOW error will result. The offending block is not read.
 r7 - points in memory to the byte following the last byte read in.
 r5L - offset from the beginning of fileTrScTab to the last (word length) entry in fileTrScTab

r1 - in case of BUFFER_OVERFLOW, r1 contains the track/sector number of the block which didn't fit and was not read in.

Destroyed: a, x, y, r0 - r10

Synopsis: LdFile is typically called from within GEOS to load a system file like the deskTop or the Preference manager. Unlike GetFile, which will load and execute an application file, LdFile always returns to its caller. In fact GetFile uses LdFile.

LdFile uses the Directory Entry to load the File Header Block which in turn contains information on where to load the file. This load information can be over-ridden by passing a 1 in bit 0 of loadOpt. In this case the address to load the file at will be taken from loadAddr.

If the file being loaded is determined to be a VLIR file, then only the first record (record 0) is loaded with the same load options available as with normal files.

GetFreeDirBlk - Get Free Directory Block

Function: Given a disk notepad page, locate a free Directory Entry on that page, or if the page is full, on the closest following page.

Called By: Utility, SetGDirEntry

Calls: GetDiskBlock, AddDirBlock

Pass: r10L - disk notepad **page number** to start looking for empty Dir. Entry
curDirHead - current **Directory Header**

Return: x - disk **error code**, 0 for no error. See Disk Errors in the Appendix.
diskBlkBuf - contains **Directory Block** containing free Directory Entry.
y - indexes from beginning of Directory Block to start of free Directory Entry in DiskBlkBuf
r10L - contains the disk notepad **page number** where the Directory Entry was found.
curDirHead - updated Directory Header. Note: curDirHead should be written back to disk at some time in case an extra Directory block was allocated.

Destroyed: a, x, y, r0 - r1, r3, r5, r7, r8

Synopsis: Each page in the deskTop disk notepad corresponds to one Directory Block in Track 18 of the disk. (Each page on the GEOS deskTop notepad holds 8 file icons. Each Directory Block also holds 8 Directory Entries.) The number passed in r10L is the notepad page on which to try and put the file (i.e., r10L = 4 means put the file on page four of the notepad), GetFreeDirBlk will try to allocate a Directory Entry on that page.

If there is no room on the requested page, GetFreeDirBlk will look on the following pages by looking in the corresponding Directory Blocks for those pages. (Directory Blocks are chained together just as data block in a file.) If necessary, GetFreeDirBlk will allocate a new Directory Block. If all 18 possible Directory Blocks are already allocated, the the routines returns an error flag.

If the Directory Entry is to be placed on a disk note pad page which hasn't been allocated yet, (e.g. the last page of the notepad is page 4 and

r10L requests page 10), then blank disk notepad pages and blank corresponding Directory Blocks are allocated until the requested page is reached.

BlkAlloc

Function: Allocates up to 127 blocks on the disk for writing a file.

Called By: SaveFile, WriteRecord

Calls: CalcBlksFree, SetNextFree

Pass:

r2 - **number of bytes** to allocate space for, up to 32,258
r6 - pointer to beginning of buffer to use as the track/sector allocation table, usually **fileTrScTab**, or fileTrScTab + 2 if the first byte in fileTrScTab is being used to point to the File Header.
curDirHead - **current Directory Header**. Use GetDirHead to read it.
interleave - desired sector interleave on disk: the number of sectors on disk to skip between sectors allocated to a file to achieve maximum efficiency. The closest interleave that the turbo routines can use effectively is 8. This value is recommended for programs and data files that are loaded in their entirety. The ReadPortion routine, however, isn't fast enough for an 8 sector interleave so files that will often be accessed with ReadPortion should be given an interleave of 9.

Return: r2 - the number of blocks allocated
r3L - Track number of the last block allocated
r3H - Sector number of last block allocated
x - error status
0 = successful
INSUFFICIENT_SPACE = not enough blocks available on disk.
curDirHead will be left with BAM bits still allocated in case of error. Make sure there are enough blocks before calling this routine.
curDirHead - BAM portion of New Directory Head was modified to allocate the newly allocated blocks. NOTE: BlkAlloc does not write curDirHead back out to disk. Use PutDirHead.
fileTrScTab - contains the track/sector table of the allocated sectors.
r8L - number of data bytes stored in last sector.
r6 -

Destroyed: a, x, y, r4 - r8

Synopsis: Call BlkAlloc to allocate space on the current disk for writing a file. BlkAlloc allocates a chained list of blocks for writing a file to the current disk. The BAM in curDirHead (the image of the Directory Header in memory) is updated and the track and sectors allocated for the file is stored in the one block table fileTrScTab, beginning with the location pointed to by r6.

BlkAlloc allocates blocks starting on track 1, and moves out toward track 35. It attempts to choose sectors at a specific interleave distance apart for optimum disk speed. Before allocating the sectors, BlkAlloc makes sure there is enough room on the disk.

The first entry pointed to by r6 within fileTrScTab is the location of the first free block. (r6 is trashed by this routine so the caller should retain r6's value if different from fileTrScTab). The second entry is the second block and so on. The track and sector of the second block is written into the first two bytes of the first block in accordance with standard c64 sector chaining practices. The track and sector of the third block is written into the first two bytes of the second block and so on. The final block doesn't need to point to a next block and so stores 0 in the first byte and the number of bytes in the block which are used.

BlkAlloc can allocate a maximum of 32,258 bytes (127 blocks) at a time. For smaller records that need to grow, use NxtBlkAlloc to allocate additional blocks to an existing file.

NxtBlkAlloc

Function: Same as BlkAlloc except that the starting sector from which to allocate blocks may be specified. This enables files to grow while maintaining optimum interleave if possible

Calls: CalcBlksFree, SetNextFree

Pass: r3 - The Track/Sector to add the interleave to and start looking for the next open block. Often this is the last sector in a previously allocated chain.

r2 - **number of bytes** to allocate space for, up to 32,258

r6 - pointer to beginning of buffer to use as the track/sector allocation table, usually within **fileTrScTab**.

curDirHead - **current Directory Header**. Use GetDirHead to read it in originally.

interleave - desired sector interleave on disk: the number of sectors on disk to skip between sectors allocated to a file to achieve maximum efficiency. The closest interleave that the turbo routines can use effectively is 8. This value is recommended for programs and data files that are loaded in their entirety. The ReadPortion routine, however, isn't fast enough for an 8 sector interleave so files that will often be accessed with ReadPortion should be given an interleave of 9.

r2 - Number of bytes to allocate space for

r6 - Pointer to beginning of a buffer to use as the track/sector allocation table for the next block of the file, usually points to **fileTrScTab**.

curDirHead - **current Directory Header**.

interleave - desired sector interleave on disk. See BlkAlloc.

Return: r2 - the number of blocks allocated
r3L - Track number of the last block allocated
r3H - Sector number of last block allocated
x - error status

0= successful

INSUFFICIENT_SPACE = not enough blocks available on disk.
curDirHead will be left with BAM bits still allocated in case of error. Make sure there are enough blocks before calling this routine.

curDirHead - BAM portion of New Directory Head was modified to allocate the newly allocated blocks. NOTE: BlkAlloc does not write curDirHead back out to disk. Use PutDirHead.

fileTrScTab - contains the track/sector table of the allocated sectors.
r8L - number of data bytes stored in last sector.

Destroyed: a, x, y, r0 - r15

Synopsis: NxtBlkAlloc is usually used to allocating new blocks on the disk to a currently existing file or record. Given a track and sector to start looking from, NxtBlkAlloc will create a new allocation table for the additional blocks, and update the BAM in curDirHead (the image of the Directory Header in memory).

Like BlkAlloc, NxtBlkAlloc can allocate a maximum of 32,258 bytes at a time. See BlkAlloc for more details.

SetNextFree

Function: Finds the next free sector on disk given the last sector and an interleave and marks it as used in the BAM (as stored in curDirHead). The sector is added to fileTrScTab.

Called By: BlkAlloc, NxtBlkAlloc, SetGEOSDisk

Pass: r3L, r3H - The **track and sector** to start looking from. If the start track in r3L is 18 then we are allocating a directory block and SetNextFree will confine itself to track 18. If the start is not 18 then track 18 will be skipped when searching for the next available sector.

curDirHead - The **Directory Header** of the current disk contains the BAM which is updated to reflect the allocated block.

interleave - **desired sector interleave** on disk: the number of sectors on disk to skip between sectors allocated to a file to achieve maximum efficiency. The closest interleave that the turbo routines can use effectively is 8. This value is recommended for programs and data files that are loaded in their entirety. ReadPortion, however, isn't fast enough for an 8 sector interleave so files that will often be accessed with ReadPortion should be given an interleave of 9.

Return: r3L, r3H - **track and sector** of allocated block
x - error status:
0 = successful
INSUFFICIENT_SPACE = not enough block available on disk.
curDirHead will be left with BAM bits still allocated in case of error. Make sure there are enough blocks before calling this routine.
curDirHead - The **BA in curDirHead is updated** to reflect the allocated block. The Directory Header is not written out to the disk, however.

Destroyed: a, y, r6, r7, r8H

Synopsis: SetNextFree finds the first free sector from a given sector with a given interleave. It marks the BAM bit as taken and returns the track/sector allocated. SetNextFree is used by BlkAlloc to allocate a series of blocks each the correct distance apart on the disk (interleave). Interleaving blocks allows the fastest possible access to a file or record.

FindBAMBit

Function: Get information for a sector on disk given its track and sector numbers

Calls: FindBAMBit

Pass: r6L, r6H - track and sector numbers

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk.

Return: Except for the zero flag, these variables are for For 1541 drives only.

a - the byte containing the bit for the block. All bits except for the sector we want to check have been masked out.

zero flag - set to reflect availability value in a:

1 = sector unused,

0 = sector used, valid for all drives

r7H - Offset into curDirHead where total # of blocks available on this track is, Don't count on for non 1541 drives.

x - offset into curDirHead/BAM to byte containing this blocks availability bit.

r8H - A bit mask for isolating the bit for this block from the BAM byte containing it.

Destroyed: nothing

Synopsis: Given an the track and sector numbers for a block, return whether or not its bit is set in the BAM. Zero Flag set if sector is unsued.

Later versions of the kernal will have more than 1 BAM -- two sided drives for example. Hence the other information returned by this routine will not be useful to a non-1541 (or 1541 clone) disk drive.

FreeBlock

Function: Free a single block in BAM.

Calls: FindBAMBit

Pass: curDirHead - Directory Header
r6L,r6H - track and sector numbers

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk.

Return: x - **disk error status**, 0 = OK, see Appendix for disk errors.
curDirHead - Directory Header minus one BAM bit.

Destroyed: a, r7H, r8H

Synopsis: Given an initialized disk drive, and a track/sector number, free a block by setting its BAM bit to 1. This is a new routine for V1.3 and beyond. The application should check the version of the GEOS Kernal it is running with before calling FreeBlock through the jump table. If this is a V1.2 disk then FreeBlock may still be called, but only by calling the address directly within the GEOS Kernal. This address is (to be supplied)

SetGDirEntry

Function: Usually called by **SaveFile**. SetGDirEntry builds a Directory Entry in dirEntryBuf and inserts it into the first empty slot in a Directory Block starting with the Directory Block whose number is passed in r10L. Note this routine is for use with GEOS format disks. (See CkDkGEOS, and SetGEOSDisk.)

Called By: SaveFile. SaveFile is probably what you want to use

Calls: BldGDirEntry, GetFreeDirBlk

Pass: r6 - Points to fileTrScTab, a chain of **allocated blocks** for the file
r9 - Points to **File Header** block as stored in fileHeader. The first two bytes of the file header when stored to disk will contain (00,FF). When this routine is called though, these two bytes should point to a null terminated string containing the filename. See the FileStructure section of this manual for more details on the File Header Block.

curDirHead - Current **Directory Header**

FileTrScTab - Contains the **track and sector allocation table** for the entire file

Return: r6 - A pointer to first unused block in fileTrScTab (for passing along to WriteFile. The first block is used by the File Header, while the second block, if this is a VLIR file is used to hold the index table.)

dirEntryBuf - contains the Directory Entry created from the File Header block including the time and date stamp and pointers to the Header Block and the Index Table block if the file is a VLIR file.

diskBlkBuf - Contains the Directory block with the new Directory Entry inserted into it.

curDirHead - May be changed by call to GetFreeDirBlk

NOTE: curDirHead should be written back to disk in case a Directory Block was added by the internal call to GetFreeDirBlk.

Destroyed: a, y, r0 - r5, r7, r8

Synopsis: SetGDirEntry (Set GEOS Directory Entry) creates a Directory Entry for a file and writes it to disk. Usually SetGDirEntry is called from within SaveFile as part of the process of writing a file to disk. BlockAlloc should

already have been called to allocate space on the disk for the file, placing this information in the variable **fileTrScTab** (a table of the chain of blocks requisitioned for the file).

Given the deskTop notepad page number on which to put the file's icon in **r10L** (i.e. **r10L** = 4 means put the file on page four of the notepad), **SetGDirEntry** will call **GetFreeDirBlk** to allocate a Directory Entry on that page. (Each page on the GEOS deskTop notepad holds 8 file icons. Each Directory Block holds 8 Directory Entries. This is no coincidence.) If there is no room on the requested page, **GetFreeDirBlk** will look on the following pages (by looking at the corresponding Directory Blocks for those pages), and if necessary, allocate a new Directory Block. If all directory pages have been used, **GetFreeDirBlk** returns an error as will **SetGDirBlk**.

GetFreeDirBlk returns the Directory Block in **diskBlkBuf**. All the information for the Directory Entry is specified in the File Header pointed to by **r9**, with the exception of the current time and date. **SetGDirEntry** calls **BldGDirEntry** to build the Directory Entry in **dirEntryBuf** from the Header-Block.

BldGDirEntry, allocates the first block in **fileTrScTab** to the Header Block. If the file is a VLIR file, the second block is allocated for the index table. **R6**, which started off pointing to the first byte in **fileTrScTab**, is left pointing to the bytes containing the track and sector of the next available block in the **fileTrScTab**. **r6** is thus incremented by 2 in the case of a regular SEQUENTIAL file and incremented by 4 in the case of a VLIR file. To put it another way, **r6** points to the track and sector of the first block available for the file's data. **GetGDirEntry** then stores \$00, FF in the first two bytes of the File Header as the File Header has no use for a next block pointer.

SetGDirEntry copies **dirEntryBuf** to **diskBlkBuf** and writes the current time and date to the File Header. It then writes the completed Directory Entry out to disk.

Note: **SetGDirEntry** does not write the FileHeader, Directory Header, or a VLIR file's Index Table back out to disk. It merely allocates the blocks in the **fileTrScTab** and updates the Directory Entry and File Header as stored in memory accordingly.

The Directory Header should be written back to disk soon after a call to **SetGDirEntry** in case a directory page was added by **GetFreeDirBlk**.

BldGDirEntry – Build GEOS Directory Entry

Function: Usually called by **SetGDirEntry** which is usually called by **SaveFile**. BldGDirEntry Creates a **Directory Entry** for a file given its GEOS File Header. Called when saving a GEOS file to disk.

Pass: r6 - pointer to allocated blocks in **fileTrScTab**
r9 - points to **File Header** block as stored in fileHeader. The first two bytes of the file header when stored to disk will contain (00,FF). When this routine is called though, these two bytes should point to a null terminated string containing the filename. See the File Structure section of this manual for more details on the File Header Block.
fileTrScTab- contains a Track and Sector chain for the blocks allocated for the file.

Return: r6 - A pointer to first unused block in fileTrScTab, (for passing along to WriteFile. The first block is used by the File Header, while the second block, if this is a VLIR file is used to hold the index table.)
dirEntryBuf - contains the Directory Entry created from the File Header block including the time and date stamp and pointers to the Header Block and the Index Table block if the file is a VLIR file.

Destroyed: a, y, r0 - r5, r7, r8

Synopsis: BldGDirEntry builds a Directory Entry in memory (at the 30 byte buffer dirEntryBuf) by using information in the file's File Header Block. This is one of the routines called when saving a GEOS file to disk. BlockAlloc should already have been called to allocate space on the disk for the file, placing this information in the variable **fileTrScTab** (a table of the chain of blocks requisitioned for the file). GetFreeDirBlk should also have been called to allocate a free Directory Entry on the disk.

BldGDirEntry allocates the first block in fileTrScTab for storing the file's Header Block. If the file is a VLIR file the second block is allocated for the index table. r6 which started off pointing to the first byte in fileTrScTab is left pointing to the bytes containing the track and sector of the next available block in the fileTrScTab. r6 is thus incremented by 2 in the case of a regular SEQUENTIAL file and incremented by 4 in the case

of a VLIR file. To put it another way, r6 points to the track and sector of the first block available for the file's data. Finally, GetGDirEntry then stores \$00,FF in the first two bytes of the File Header. (See the description of the file header block for why these values are written.) BldGDirEntr returns a completed Directory Entry in dirEntryBuf.

FollowChain

Function: Given the first block in a track/sector chain, build a track/sector table for the file.

Pass: r1l, r1H - track/sector of **first block** in chain
r3 - pointer to buffer in which to store **track/sector table**, usually **fileTrScTab**.

Return: r1l, r1H - Track/sector of **last block** in chain
r3 - Pointer to beginning of **track/sector table**.
diskBlkBuf - The last block in the file.

Destroyed: a, y, r1, r4

Synopsis: Given a pointer to a chain of sectors stored on disk, FollowChain builds a track/sector table in memory. A track/sector table is just a list of words containing the track and sector of each block in a file. fileTrScTab is usually used to store this information.

FastDelFile

Function: Deletes a file in a single disk access when a list containing the track/sectors of every block in the file is already available

Calls: GetDirHead, FreeBlock, PutDirHead

Pass: r0 - pointer to null terminated filename
r3 - pointer to T/S list in **fileTrScTab**
curDrive - the device name of current disk drive. Look for file on current disk.
fileTrScTab - the list containing the track and sectors of each disk sector allocated to the file.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to **SetDevice**. Thus **NewDisk** or **OpenDisk** must have been called originally. **NewDisk** and **OpenDisk** also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Return: x - **disk error status**, 0 = OK; see Appendix for disk errors.
curDirHead - **Directory Header/BAM** changed in memory and written back to disk.
x - Error status, 0 = OK, see Appendix for disk errors.
turbo - Turbo code turned off but not purged from drive
dirEntryBuf - Returned from call to **FindFile**, the **Directory Entry** for the deleted file remains in memory.

Directory Block - The Directory Block containing the Directory Entry for the file is written back out to disk.

File Header Block/VLIR Index Table - Removed from disk along with rest of file.

Destroyed: a, y, r0 - r9

Synopsis: DeleteFile only needs the filename to delete the file, but it does many disk accesses and is slow. Often when dealing with a file, more

information than just the filename is available. If a track/sector list of blocks allocated for the file is available in **fileTrScTab**, **FastDelFile** may be used. It only needs one disk access to update all the BAM bits necessary for deleting the file. The track/sector list should be in **fileTrScTab** as built by **BlkAlloc** or **ReadFile**. See description of track/sector table.

FreeFile

Function: Given the Directory Entry for the file, free all BAM bits corresponding to sectors in the file.

Called By: DeleteFile

Calls: GetDirHead, PutDirHead

Pass: dirEntryBuf - Directory Entry for the file.
r9 - points to dirEntryBuf.

Initialized Drive - In order to access the disk at all, it must have been initialized: its **Disk ID** must have been read into the drive and the GEOS variable **curDrive**, the device number of the **current drive**, must have been set with a call to SetDevice. Thus NewDisk or OpenDisk must have been called originally. NewDisk and OpenDisk also set other variables. Depending on what routines have been called since the current disk was initialized, disk accesses can often be optimized by substituting a more specific/primitive routines when their parameters are already in memory. Careful planning can speed up disk accesses.

Return: x - **disk error** status, 0 = OK. See Appendix for disk errors
Directory Header/BAM - curDirHead is updated to indicate the newly freed blocks and written out to the Directory Header/BAM block on disk.

Directory Block - The Directory Block containing the Directory Entry for the deleted file is **not** updated.

File Header Block/VLIR Index Table - Removed from disk along with rest of file.

Destroyed: a, y, r0 - r9

Synopsis: Given a file's Directory Entry, FreeFile clears all BAM bits for sectors allocated to that file, and Updates the BAM in the Directory Header. Works for all types of files including VLIR files. In the case of VLIR files, the index table is removed and each record is deleted. The file's Directory Entry and the Directory Block that contains it are not removed/updated. That is done by DeleteFile. FreeFile is the workhorse routine called by DeleteFile.

16.

Primitive Routines

In Chapter 15, we covered the Intermediate-Level GEOS routines. We will now consider some of the Primitive routines which made up GEOS; they include:

- InitForIO
- DoneWithIO
- PurgeTurbo
- EnterTurbo
- ReadBlock
- WriteBlock

InitForIO

Function: Call before using serial bus; disable interrupts and sprites; set up memory map for c64 Kernal and I/O banks in; and set up some serial transfer bytes.

Pass: nothing

Return: Interrupts disabled, sprites disabled, and status saved, forces NMI to occur once, but remains disabled afterwards. IRQ off, c64 Kernal and I/O in.

Destroyed: a

Synopsis: Call InitForIO before any action accessing the serial bus. Anything that can steal cycles away from the processor like interrupts or sprite DMA is turned off. The present configuration of the memory map (which banks are swapped in and which are out) is saved and the c64 Kernal and I/O space is swaped in. Dummy IRQ and NMI vectors are loaded which do nothing. The serial port is set to its normal configuration. A counter is set to count down and cause an NMI so that the NMI line will stay low till released in DoneWithIO. Raster interrupts are turned off. The sprite enable register is saved, sprites turned off and any sprite DMA is allowed to completely finish.

DoneWithIO

Function: Call when done with the serial bus. Enables interrupts, sprites, restores memory map and resumes normal operation.

Pass: nothing

Return: System restored to configuration before InitForIO was called.

Destroyed: a

Synopsis: Call DoneWithIO to reset the system to the same configuration it had before InitForIO was called. Sprites are turned back on, as are NMIs and IRQs and the raster interrupts. The old configuration of the memory map is restored.

PurgeTurbo

Function: Returns control of the 1541 to the ROM resident DOS and sets flag indicating Turbo code no longer resident in the drive.

Called By: GetBlock, PutBlock, NewDisk, ReadFile

Calls: InitForIO, DoneWithIO

Pass:
curDrive - the number of the drive on which to indicate removal of the Turbo code
turboFlags - the turbo flag for the current disk, either turboFlags for drive 8 or turboFlags+1 for drive 9
Bit 7 - 1 if turbo software resident in the drive's RAM
Bit 6 - 1 if turbo software is running on the 1541

Return:
x - unchanged, if x was error status, 0 = OK
turboFlags - the turbo flag for the current disk, either turboFlags for drive 8 or turboFlags+1 for drive 9
Bit 6 - reset to 0. Turbo software not running on the 1541

Destroyed: a, x, y, r0 - r3

Synopsis: Purge Turbo is called turn off the execution of Turbo code and mark the turboFlags bit for the current drive indicating that the Turbo code is no longer resident in the disk drive. Call PurgeTurbo when executing a 1541 DOS command which may have affected the RAM in the drive.

The Turbo code is normally not removed from the drive between accesses so that EnterTurbo need not re-transfer the code up to the drive. All that is necessary to restart the turbo. Usually, all necessary calls to Enter/Exit Turbo are made by the higher level file and disk routines.

EnterTurbo

Function: Runs the disk Turbo code on the current drive, transferring the code if necessary.

Called By: GetBlocks, NewDisk

Calls: SetDevice, InitFor Turbo, DoneWithTurbo,

Pass: curDrive - the **number of the drive** on which to execute the Turbo code.

turboFlags - the turbo flag for the current disk, either turboFlags for drive 8 or turboFlags+1 for drive 9.

Bit 7 - 1 if turbo software resident in the drive's RAM.

Bit 6 - 1 if turbo software is running on the 1541.

Return:

x - disk error status, 0 = OK

Destroyed: a, y

Synopsis: EnterTurbo causes the 1541 intelligent drive to start executing the disk turbo code. If the turbo code is not resident in the drive, it is transferred there. Usually, EnterTurbo must be called every time a disk operation is desired because it is not left running on the drive. The Turbo software will grab and hold the serial bus. In order for any other device to use the serial bus, ExitTurbo must be called. This is done by all the higher level file and disk routines. If it is necessary for an application to use the disk at this low level, it is usually not a good idea to leave the Turbo software active on the 1541 and holding the bus. It is a good idea then to call ExitTurbo after the disk operation is completed.

ReadBlock

Function: Same as GetBlock but assumes turbo code is turned on and interrupts and sprites are off.

Called By: GetBlock

Pass: r1L,r1H - track/sector of block to read .
curDrive - the number of the current drive, either 8 or 9.

System: The drive must have been initialized via OpenDisk, or NewDisk, and must be the selected device on the serial bus, via a call to SetDevice. The c64 I/O space should be bank switched in and interrupts and sprites disabled.

Return: x - error status, 0 = OK, see Appendix for disk errors
r4 - pointer block read in (unchanged)
r1 - unchanged

Destroyed: a, y

Synopsis: ReadBlock is the most primitive disk accessing routine in GEOS. When reading and writing long chains of blocks to disk, it is desirable to minimize overhead time. The higher level routines provided for reading and writing chained links of blocks to the disk are pretty much optimized. Some speed up may be possible in writing files by doing the write of all the blocks and then doing the verify afterwards. Another speed up may be possible if your application uses its own custom data structure which is not based on a linked chain. Use ReadBlock within the following context.

jsr EnterTurbo	;transfer and start turbo code running
jsr InitForIO	;turn off interrupts, sprites, get I/O space
jsr ReadBlock	
...	;read some blocks
jsr ReadBlock	
Jsr DoneWithIO	;re-enable sprites, interrupts, etc.

Pass track and sector of the block you want. ReadBlock will load a sector (block) from the current 1541 drive into c64 memory at the requested address. After being read, the block still contains the track/sector pointer which takes up the first word in the sector on disk.

For example, suppose your application stores key information as the first few bytes of the first block in each record of a VLIR file. After you have the Index Table (in fileHeader perhaps), you may read the first block of each record using the track and sector from the Index Table to retrieve the key info bytes you need.

WriteBlock

Function: Same as PutBlock but assumes turbo code is turned on and interrupts and sprites are off.

Called By: PutBlock

Pass: r1L,r1H - track/sector of block to read
r4 - pointer to buffer holding data to write to disk, usually diskBlkBuf.

System - The drive must have been initialized via OpenDisk, or NewDisk, and must be the selected device on the serial bus, via a call to SetDevice. The c64 I/O space should be bank switched in and interrupts and sprites disabled

Return: x - error status, 0 = OK, see Appendix for disk errors
r4 - pointer block read in (unchanged)
r1 - unchanged

Destroyed: a, y

Synopsis: WriteBlock is the disk primitive that writes a block from the c64 memory to the currently open 1541. It requires that interrupts, and sprites are off and the disk turbo is on. For a file to be stored correctly, blocks must be linked, that is, the first two bytes of each block while still stored in c64 RAM should contain the track and sector of the disk location to store the next block at. The block to write is usually stored in diskBlkBuf. Use ReadBlock within the following context.

```

jsr EnterTurbo           ;transfer and start turbo code running
jsr InitForIO            ;turn off interrupts, sprites, get i/o space
jsr WriteBlock
...                      ;read some blocks
jsr WriteBlock
jsr DoneWithIO           ;re-enable sprites, interrupts etc.
```

Pass track and sector of the block you want to write to. While still in memory the block should already **contain the track/sector** pointer to the next block on disk.

Usually one of the higher level file routines is what is needed. Standard disk errors are returned.

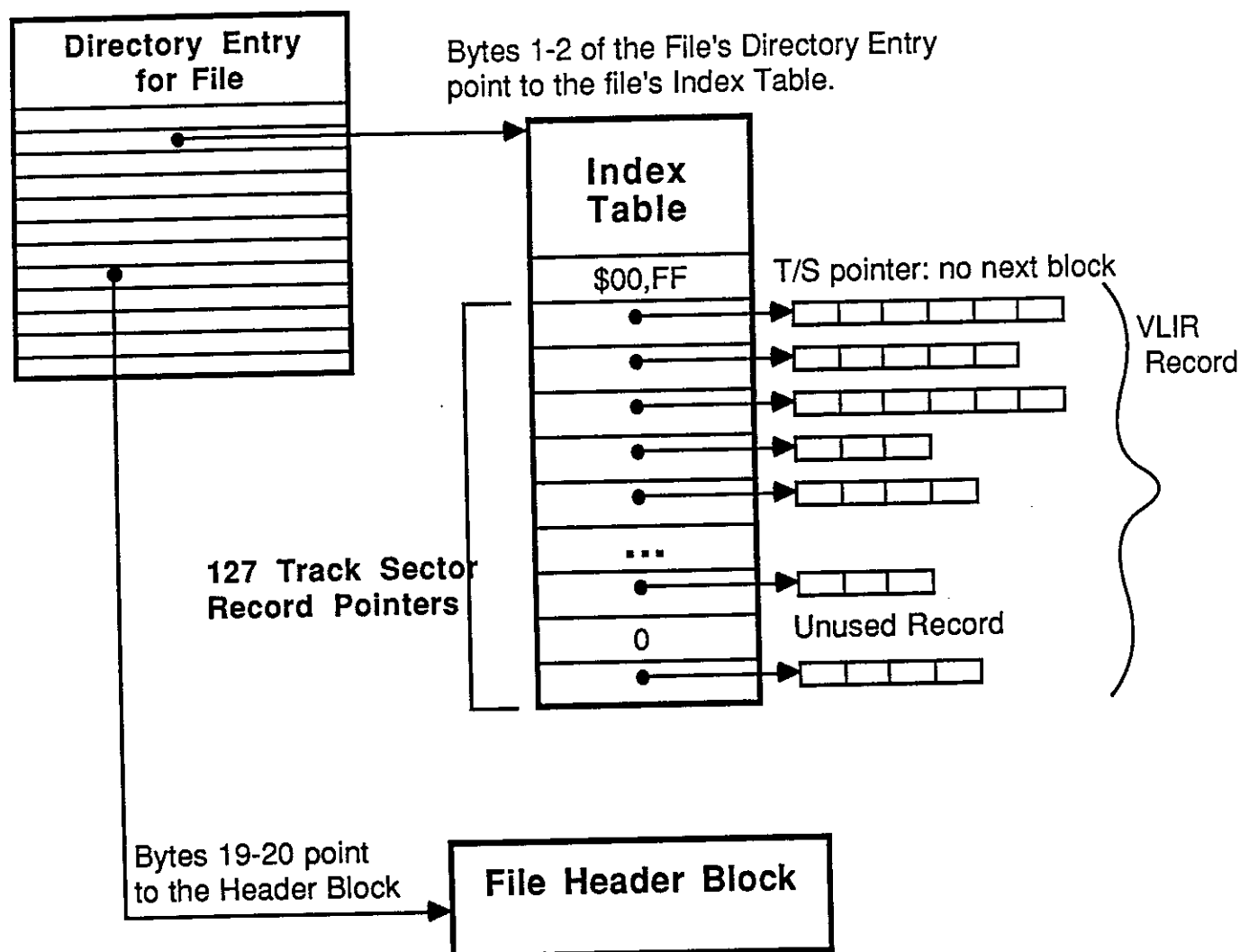
VLIR Files

The VLIR file structure was created to allow applications to grow much larger than the 30k available to them in GEOS. With a faster 1541 disk speed, it becomes practical to break an application up into several different modules, and swap them in as needed. A good way to organize such an application is to keep one module always resident while the others share a common memory area. The resident module is allowed to call subroutines in any of the other swap modules but the other modules may only call routines in the resident module. This keeps the application from getting bogged down with endless swapping. Applications tend to execute out of one module for a while, and then swap modules and execute out of another for a while.

A VLIR file is comprised of several modules referred to as records. Each record, is a chained link of blocks just like a regular Commodore file. Thus a VLIR file is somewhat like a collection of files. The same routines used to save a regular SEQUENTIAL file to disk may be used to save individual records in a VLIR file. In addition, several VLIR specific routines are provided.

The VLIR file routines allocate sectors on disk for records the same as is done for regular files, using the one block track/sector allocation table, fileTrScTab. Each record may therefore be from 0 to 127 blocks long, (just under 32k: 32,258 bytes), the maximum number of track/sector pointers fileTrScTab can hold. If the application uses the background screen buffer for program space, it has the use of memory from \$400 to \$8000 which is also just under 32k. An Index Table, holds the track/sector pointers to the first block in each record. The diagram below shows how the VLIR file uses an Index Table to organize the records in the file.

VLIR is an acronym for Variable Length Indexed Record. Both applications and data files may be stored in VLIR file. For example, the font files are divided into several records, one for each point size.



VLIR - Variable Length Indexed Record File Structure

A VLIR file can be identified by looking at the GEOS Structure type byte in the file's Directory Entry. In addition, the Directory Entry contains a track/sector pointer to the file's Index Table. In a regular SEQUENTIAL file this word usually point to the first data block in the file. See the beginning of the file system section for more details on the Directory Entry structure. The Index Table consists of 127 entries, numbered 0 to 126, where each entry is a pointer to a record. The rest of the entries in the Directory Entry, such as the pointer to the Header Block, are the same.

VLIR Routines

The routines for reading and writing records, closely resemble those one might expect for manipulating objects in a linked list: NextRecord, PreviousRecord, and others.

This "linked list" concept makes use of a pointer to the current record. This pointer may be set directly or set to the next or previous record. The current record may be deleted, read from, or written to. At each access, the full record must be dealt with. Thus the application should provide sufficient RAM at any one time to accomodate the largest possible record it could be processing. New empty records may be inserted before, or appended after the current record. New Records are empty and may be written to. Presently there is no way to detach a record and re-attach it somewhere else. DeleteRecord is destructive, i.e., frees up the sectors, and InsertRecord only works with empty records.

The Index Table may be stored in memory, often in the fileHeader buffer, to make it possible to go directly to a record using PointRecord instead of advancing one record at a time with NextRecord or PreviousRecord.

An attempt has been made to return meaningful error flags concerning operations on the structure. The following is a list of possible errors as returned in the x register by VLIR Record routines.

Error Messages

UNOPENED_VLIR_FILE

This error is returned upon an attempt to Read/Write/Delete/Append a record of a VLIR file before it has been opened with OpenRecordfile.

INVALID_RECORD

This error will appear if an attempt is made to Read/Write/Next/Previous a record what doesn't exist (isn't in the Index Table). This error is not fatal, and may be used to move the Record pointer to the end of the record chain.

OUT_OF_RECORDS

This error occurs when an attempt is made to Insert/Append a record to a file that already contains the maximum number of records allowed (127 currently).

STRUCT_MISMATCH

This error occurs when a routine supporting a function for one type of file structure is called to operate on a file of different type.

Creating a VLIR File

Use the SaveFile routine to initially create a VLIR file. The File Header should contain the following values:

c64 File Type	- USER
GEOS File Structure Type	- VLIR
For Data Files:	
Start Address:	0
End Address:	FFFF (-1)
For Applications:	
Start Address:	Location to load the first record when the application is loaded.
End Address:	The Start Address - 1. This causes an empty VLIR structure to be created by SaveFile.

This creates a VLIR file on disk with an Index Table with no records. The current Record pointer is set to -1: a null pointer. Before any manipulation of the file is possible, it must be opened with OpenRecordFile. This loads certain internal buffers GEOS needs. With a completely empty Record file like this, the first record must be created with AppendRecord. After that calls to InsertRecord, AppendRecord, and DeleteRecord are possible.

When through with the file, it is imperative that the programmer close it by calling CloseRecordFile. This will update the file's Index Table, the disk BAM, and the "blocks used" entry in the file's Directory Entry. Note that at present only one VLIR may be opened at a time.

A description of the routines available specifically for VLIR files appears below.

OpenRecordFile

Function: Open an existing VLIR file for access given its filename.

Calls: FindFile, GetBlock

Pass: r0 - pointer to null-terminated filename

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return:

- x - disk error status (0 =OK), see Appendix for disk errors.
- fileHeader - **IndexTable** stored in fileHeader block.
- usedRecords - **number of Records** in file.
- fileWritten - flag indicating if file has been written to since last change to BAM or IndexTable. Zero = no change yet.
- curRecord - Zero if at least one record in file, else set to -1 for empty structure
- dirEntryBuf - **Directory Entry** for file.
- curDirHead - The **Directory Header** of the disk.

Destroyed: a, y, r1, r4 - r6

Synopsis: OpenRecordFile sets up the RAM variables above as expected by the ReadRecord and WriteRecord routines. OpenRecordFile calls FindFile to check the disk for the file. If found, the values for several variables are retrieved and the file is error checked to make sure it is a VLIR file.

CloseRecordFile

Function: Update the VLIR file's IndexTable and the disk BAM. Indicate no open VLIR file.

Calls: UpdateRecordFile

Pass: **usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf** - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized **Drive** - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: IndexTable - updated
BAM - updated
fileWritten - reset to 0,
Directory Block- if record was modified, update the **Blocks Used** entry in theDirectory Entry. The **time/date** variables in the Directory Entry are updated from the year, month, day, hour, minutes, seconds variables in RAM.

Destroyed: a, y, r1, r4, r5

Synopsis: Calls UpdateRecordFile to update the Record variables mentioned above. If the file has changed since the last write, the time/date stamp in the Directory Entry is updated. An internal GEOS variable is set to indicate no presently open VLIR files.

UpdateRecordFile

Function: Update the VLIR file's IndexTable, disk BAM, and Time/date stamp.

Called By: CloseRecordFile

Calls: GetBlock, PutBlock

Pass: **usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf** - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: Index Table - updated
BAM - updated
fileWritten - reset to 0
Directory Block - update the **Blocks Used** entry in the Directory Entry.
The **time/date** variables in the Directory Entry are updated from the year, month, day, hour, minutes, seconds variables in RAM.

Destroyed: a, y, r1, r4, r5

Synopsis: UpdateRecordFile updates the Record variables mentioned above. If the file has changed since the last write, the time/date stamp in the Directory Entry is updated.

PreviousRecord NextRecord PointRecord

Function: Adjust current record pointer to the previous, next, or to a specific record in the VLIR file.

Pass: a - Contains the **record number** for PointRecord. Not used for Next or PreviousRecord.

usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: curRecord - current **record number**
x - **error status**,
 0 - pointing to previous record
 nonzero - location didn't exist. pointer unmoved.
y - **empty flag**: zero if no error but record is empty, else nonzero.
 The actual value loaded into y is the track of the record as stored in the index table. This is zero if the record contains no blocks, i.e., nothing to point to.
r1L, r1H - The track and sector of the first block in the record used-Records,
fileHeader - unchanged

Destroyed: nothing

Synopsis: PreviousRecord, PointRecord and NextRecord adjust the current record pointer to point to a new record. You must pass a record number to point at to PointRecord. If some error occurred in moving the pointer to a new record, such as calling NextRecord when already pointing to the last record, then the error condition is returned in x and the current record pointer is unchanged. If there was no error in reading the disk, the track and sector number of the first block of the requested record is retrieved from the index table and loaded into r1. r1L, the track, is copied to y. A track of 0 indicated an empty record.

DeleteRecord

Function: Deletes current record and leaves curRecord pointing to the following record.

Calls: GetDirHead (probably redundant)

Pass: **usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf** - These variables initialized by call to OpenRecordFile. file Header contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: x - **error status**, 0 = OK
Current Record - left pointing to **following record**, or last record in list.

Destroyed: a, y, r0 - r9

Synopsis: The Current Record is deleted and curRecord is left pointing at the following Record. If the deleted Record was the last Record in the VLIR file, then the CurrentRecord pointer is left pointing at the new last Record.

WriteRecord

Function: Writes contents of memory area out to Current Record.

Calls: GetDirHead, WriteFile, BlkAlloc

Pass: r2 - **Number of bytes** to write
r7 - **Beginning address** of data block in RAM to write to disk

usedRecords, curRecord, fileWritten, fileHeader, curDirHead, dirEntryBuf - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: x - Disk **error status**, 0 = OK. See Appendix for disk errors.
fileHeader - Contains the updated **Index Table**
fileSize - Current **size in blocks** of record.
fileWritten - Loaded with \$FF indicating file modified since opening.
fileTrScTab - Table of sectors used to store the record.
disk - Old Record deleted, **new Record written.**
r8L - number of data bytes stored in last sector.
r3 - Track and sector of **last block** allocated
curDirHead - **BAM** portion of Directory Header was modified to reflect allocation of new blocks to the record. Directory Header not written to disk. Use PutDirHead for that.

Destroyed: a, y, r0 - r9

Synopsis: The Current Record on disk is deleted (blocks freed in BAM) and the contents of the indicated block of RAM is written out. **Note:** The old version of the file is deleted before the new one is written. If there is not enough room on the disk to write the new version of the file, the old file will be lost. Use CalcBlocksFree to make sure the file can be written. The time/date stamp is not updated until the file is closed.

ReadRecord

Function: Reads Current Record into memory.

Calls: ReadFile

Pass: r2 - max **number of bytes** expected. If the Current Record contains more than this, a BUFFER_OVERFLOW error will be generated.
r7 - **beginning address** of data block in RAM to read into.
usedRecords, curRecord, fileWritten,
fileHeader, curDirHead, dirEntryBuf - These variables initialized by call to OpenRecordFile. fileHeader contains index table.

Initialized Drive - curDrive, curDevice set via SetDevice and the BAM for the current disk read into the drive via OpenDisk, or NewDisk

Return: x - Disk **error status**, 0 = OK. See Appendix for disk error listing.
r7 - Pointer to byte following **last byte** read in (if non-empty record)
a - **EmptyFlag**: 0 if record was empty.
fileTrScTab - Track/sector table for the record.
r5L - Points to the **last (word length) entry** in fileTrScTab. The value of this word is [\$0, index to last data byte in block]
r1 - In case of BUFFER_OVERFLOW error, r1 contains the track/sector number of the **block which didn't fit**, and was not read in.

Destroyed: a, y, r1 - r4

Synopsis: The Current Record is read into memory at the indicated location. The fileTrScTab is built. The first two entries in fileTrScTab are identical: both store the track/sector of the first data block in the record.

Printer Drivers

This chapter is intended for

1. programmers who want to use GEOS printer drivers with their applications,
or
2. programmers who want to write a GEOS printer driver for a previously unsupported printer.

The State of Printers

There is such a multitude of different printer types on the market today that a several books could be written about their operation. In fact, several have. To find out about a specific printer or interface card consult the operator's manual or visit the local computer store.

There are two basic categories of printers: "character" (typewriters, daisywheel, band printers, etc.), and dot-matrix printers. Character printers are only capable of printing character shapes that are physically on the print wheel (band, ball, or hammers). In general, this makes them unsuitable for use with GEOS since GEOS stores and prints both character fonts and graphics as a bit map. GEOS does support a near letter-quality print mode for the 1526 Commodore printer, but to use GEOS as it was intended to be used requires a dot-matrix printer.

Dot-matrix printers are constructed with vertical lines of pins which can be individually controlled to strike the ribbon (or squirt the ink, in the case of an ink-jet printer, which also falls into the dot-matrix category) onto the paper. The device holding these pins is called the printhead. As the printhead moves across the page, different dot-columns are printed, leaving a two-dimensional pattern (matrix) of dots. Individual characters are patterns of adjoining dots on the page (see Printers Figure 1).

Printers Figure 1

**PRINTHEAD
CHARACTER MATRIX**

<p>○</p> <p>○</p> <p>○ Sweeps across</p> <p>○ horizontally</p> <p>○ to print:</p> <p>○</p> <p>○</p>	<p>---○---</p> <p>--○-○--</p> <p>-○---○-</p> <p>○○○○○○○</p> <p>○-----○</p> <p>○-----○</p> <p>○-----○</p>	<p>or</p> <p>○-----</p> <p>○○-----</p> <p>-○-----</p> <p>-○-----</p> <p>-○-----</p> <p>○○○-----</p>
---	--	---

ASCII and Graphic Printing

Dot matrix patterns usually operate in two modes. In the first, ASCII mode, an application feeds the printer ASCII character codes and the printer prints from its own internal character set. In its own memory it stores the dot pattern for all the letters. In addition to this first mode there is the ability to send the printer the actual dot patterns to print.

The printer's internal character set is used for draft and near-letter quality (NLQ) modes of printing. In draft mode the application passes the printer driver a string of regular ASCII (not Commodore ASCII) characters. The printer prints these out in its fast single strike draft mode using its internal character set. NLQ mode is just like draft mode except that several overstrikes or other methods are used in order to make the printed output sharper.

GEOS uses the graphics mode of the printer for all graphic and most text printing. This is how it is possible to print different fonts. This mode is variously referred to as Graphics Mode, Bit-Image Mode, or APA (All Points Addressable) Graphics Mode. This mode interprets bytes in the print buffer not as ASCII characters, but as bit patterns (vertically oriented) for the printhead to print. Figure 2 below shows an example of how a typical printhead might be addressed in graphics mode. Each pin on the printer is assigned a bit. The "Dot Columns as Printed" columns show the value passed to the printer and the image it produces.

Bit Value	Printhead	Dot Columns as Printed															
		01	02	04	08	10	20	40	80	AA	55	00	3C	42	81	81	42
\$01	o	o	-	-	-	-	-	-	-	-	o	-	-	-	o	o	-
\$02	o	-	o	-	-	-	-	-	-	o	-	-	-	o	-	-	o
\$04	o	-	-	o	-	-	-	-	-	-	o	-	o	-	-	-	-
\$08	o	-	-	-	o	-	-	-	-	o	-	-	o	-	-	-	-
\$10	o	-	-	-	-	o	-	-	-	-	o	-	o	-	-	-	-
\$20	o	-	-	-	-	-	o	-	-	o	-	-	o	-	-	-	-
\$40	o	-	-	-	-	-	-	-	-	-	o	-	-	o	-	-	o
\$80	o	-	-	-	-	-	-	-	o	o	-	-	-	-	o	o	-

Dot Matrix Printer Types

There are two general categories of printheads around today: 9-pin and 24-pin. 9-pin printheads use the top 7 or 8 pins to actually print in graphics mode. The bottom one or two pins are used to print descending characters. These are ASCII characters like "g" and "p" that have tails below the printline. Whether 7 or 8 pins are used to print graphics is also

dependent on the printer itself. Bit 0 may be at either the top or the bottom pin, depending on the individual printer. Since 8 bit data is easier for an 8 bit computer to handle than 7 bit data, having to spoon feed a printer 7 bit wide data can be tedious. As a bit of foreshadowing let us mention this will be discussed in more detail later when we discuss the print algorithms. Presently we continue with a general printer description.

Typically, the pins make a $1/72$ " x $1/72$ " dot, spaced $1/72$ " apart vertically. Dot-columns are spaced at $1/60$ ", $1/72$ ", $1/80$ ", or even closer depending on the printer and the mode in which it is running. 24-pin printers work basically the same way the 9-pin printers do, except at a higher resolution (24 pins in the same area as the 9 and a correspondingly higher horizontal resolution).

Printers enter and exit graphics mode one of two ways: some are given a command to enter graphics mode and stay that way until a command is given to exit graphics mode. Others are given the command to enter graphics mode, followed by a byte count. Until the count reaches zero, every byte that the printer sees is printed out in graphics mode.

Once the program is capable of individually firing pins on the printhead, the only thing preventing it from printing a whole page of solid graphics is the control of how far the printer line-feeds when told to do so. Fortunately, every printer that has a graphics mode, also has the ability to be told how far to advance the paper when a LF is encountered. The first step understanding printing in either ASCII or graphics mode is to learn how to communicate with the printer. Most printing is done through the c64's serial port. An exception to this is geoCable by Berkeley Softworks which allows you to run any centronics parallel printer from the user parallel port with GEOS. The following section deals with the c64 serial bus interface to the printers.

Talking to Printers

This section describes the way the serial bus works, the routines in the c64 Kernal ROM used to communicate with peripheral devices, and the types of interfaces available for parallel input printers.

The c64 communicates with its peripheral devices (disk drives, printers, etc) over a serial bus. The serial bus supports up to five devices connected at once in a daisy-chain fashion. There are three basic types of activity on the serial bus, "control", "talk", and "listen". The c64 is the controller of the bus, and can tell peripheral devices when to "talk" (to output

data onto the bus) or when to "listen" (accept input from the bus). The devices are assigned unique addresses which are output on the bus when a control signal from the c64 is sent out. These "addresses" are single byte numbers based on device type. All serial printers are assigned the number 4. To work with the c64, a printer must recognize a 4 on the serial bus as its "address" and react to the next byte which is one of several possible command bytes. It can be any valid command byte that the device recognizes. This second byte is called the secondary address. For more information on the serial bus and how it works, see the Commodore 64 Programmers Reference Guide (pp 362-366).

The c64 Kernal ROM has routines resident within it to operate the serial bus. These routines "talk", "un-talk", "listen", "un-listen", send secondary addresses, and receive and send data on the serial bus. These routines are called with device addresses (if needed for the routine) in the accumulator, and return error codes in the accumulator. The Kernal routines set the carry flag to indicate that the value in the accumulator is a valid error code and not just left over garbage. These primitive routines are used by printer drivers to set up transmission of data over the serial bus to the printer. For more information on the Kernal ROM routines, see the Commodore 64 Programmers Reference Guide (pp 270-304).

Parallel Interface Questions

Since many of the higher quality printers available are not equipped with interfaces for the Commodore serial bus (most have Centronics parallel interfaces), the user must either use the geoCable printer cable and geoCable printer drivers, or use a serial-to-parallel interface that recognizes the Commodore serial bus protocol and the Centronics standard. Fortunately, a few such devices exist, and are readily available to the consumer at major retailers. Some of these are: Cardco G-Whiz, Cardco Super-G, and Telesys Turboprint CG.

GEOS Printer Drivers

Now that we have covered the basics of printer operation we proceed to printer driver operation. In order for all applications to be able to talk to all printer drivers, two things were necessary.

1. All applications must see a single general interface standard.
2. A driver must be written for each functionally different printer that takes takes the application's output, and tailors it to a specific printer.

The application is responsible for one half of the work and the printer driver for the other half.

The Interface – For Graphic Printing

Printer drivers and applications pass data through a 640 byte buffer. This buffer is sized to hold eight scanlines of 80 bytes per scanline resolution. This is the maximum line width supported by GEOS. (Some applications may not support the entire width of a GEOS page. For example, geoWrite only supports 60 bytes across. In this case the application must put out blank bytes on either end of the buffer line.)

What this amounts to is the application assembles a buffer of graphics data in hi-res bitmap mode card format, and calls a printer driver routine that reorganizes the data and sends it over the serial bus. The applications programmer must then know how to format the data, and what routines in the printer driver to call. The printer driver author must implement the standard set of routines to print on a specific printer. This means reordering the bytes significantly since the printer expects bytes that represent vertical columns of pixel data while each byte of data passed in the 640 byte buffer represents eight horizontally aligned pixels. This work is done in four separate callable routines.

GetDimensions: return the dimensions in Commodore screen cards of the page the printer can support.

InitForPrint: called once per document to initialize the printer. Presently only used to set baud rates.

StartPrint: initialize the serial bus at the beginning of every page, and fake an opened logical file in order to use c64 Kernal routines to talk to the printer.

PrintBuffer: print the 640 byte buffer just assembled by the application when printing in graphics mode.

StopPrint.: do end of page handling, a form feed and for 7 bit graphics printing, flush the remaining scanlines in the buffer.

The application is in control of the printing process. It calls `InitForPrint` once to initialize the printer. Then `StartPrint` is called to set up the serial bus. After that `GetDimensions` is usually called to find out the width of the printable line and the max number of lines in the page. The application then fills the buffer with bit map data in card format and calls `PrintBuffer` is to print it. As soon as a full page has been printed, `StopPrint` is called to perform the formfeed and any other end of page processing necessary. The process begins again next page begins with a `StartPrint`.

ASCII Printing

All ASCII printing is done on a 66 lines per page and 80 character per line basis. The application passes the printer driver a null terminated ascii string. Any formatting of the document such as adding spaces to approximate tabs should be done by the application. All end-of lines are signaled by passing a carriage return to the driver. The driver will output a CR as well as a linefeed for every CR it receives in order to move the printhead to the beginning of the next line. For some applications, such as `geoPaint` a draft or NLQ mode of printing do not make sense. Others such as `geoWrite` will offer draft and NLQ modes of printing for printing text and will skip any embedded graphics in the document.

The procedure for ASCII printing is much the same as for graphic printing. The application calls `InitForPrint` once to initialize the printer. If NLQ mode is desired then `setNLQ` is called. The application then calls `StartASCII`, instead of `StartPrint` to set up the serial bus. The application may now begin sending lines. It passes a null terminated string of characters, pointed to by `r0`, to `StartASCII`. Spaces used to format the output should be embedded within the string passed to `StartASCII`. A carriage return should be printed at the end of every line.

StartASCII: same as `StartPrint` except for printing in draft or nlq modes.

PrintASCII: Use this routine instead of `PrintBuffer` for draft and NLQ printing. The application passes a null terminated ASCII character string to the driver instead of the 640 byte buffer, and the printer prints in its own charset.

SetNLQ: Send the printer whatever initialization string necessary to put it into near letter quality mode.

Calling a Driver from an Application

Printer drivers are assembled at PRINTBASE (\$7900), and may expand up to \$7F3F. Applications must leave this memory space available for the printer driver. In addition, the Application must provide space for two 640 byte RAM buffers. The application uses the first buffer to pass the 80 cards (640 bytes) of graphics data to the driver. The driver uses the other internally. These two buffers pointed at by r0 and r1 when a driver routine is called.

At the beginning of each printer driver is a short jump table for the externally callable routines. An application calls routines in the printer driver by performing a jsr to the routine's entry in the jump table.

Print Driver JumpTable

```
.psect PRINTBASE

InitForPrint:
    ;first routine at PRINTBASE
    jmp     r_InitForPrint
StartPrint:
    ;second routine at PRINTBASE + 3
    jmp     r_StartPrint
PrintBuffer:
    jmp     r_PrintBuffer                ;third routine at PRINTBASE + 6
StopPrint:
    ;fourth routine at PRINTBASE + 9
    jmp     r_StopPrint
GetDimensions:
    jmp     r_GetDimensions              ;fifth routine at PRINTBASE + 12
PrintASCII:
    jmp     r_PrintASCII                 ;sixth routine at PRINTBASE + 15
StartASCII:
    ;7th routine at PRINTBASE + 18
    jmp     r_StartASCII
SetNLQ:
    ;8th routine at PRINTBASE + 21
    jmp     r_SetNLQ
```

Typically, an application will go through the following procedure to print out a page.

Using a Printer Driver from an Application

For Graphics Printing:

(A) Call `GetDimensions (PRINTBASE+12)` to get: (1) the length of the line supported by the printer (constant is `CARDSWIDE`) usually 80 but sometimes 60, in x, and (2) the number of rows of cards in a page (which is the same as the number times to call the `PrintBuffer`) in y (constant is `CARDSDEEP`).

(B) Call `InitForPrint (PRINTBASE)` once per document to initialize the printer. Call `StartPrint (PRINTBASE + 3)` once per page to set up the commodore file to output on the serial bus. Any errors are returned in x and the carry bit is set. If no error was detected, x is returned with 0.

(C) To print out each row of cards (do 1,2, and 3 for each line) do the following. (1) Load a 640 byte buffer with a line of data (80 cards) and load r0 with the start address of the 640 byte buffer. (2) Load r1 with the start addr of 640 bytes RAM for the print routines to use. Load r2 with the color to print. Multicolor printers require several passes of the print head. Each in a different color, each with a different set of data. For each line then, `printBuffer` is called for each color. (3) Call the `PrintBuffer` routine (`PRINTBASE + 6`). NOTE: r1 must point to the same memory for the whole document, and must be preserved between calls to `PrintBuffer`. r0 can change each time `PrintBuffer` is called. Goto 1 the until page is complete.

(D) Call the `StopPrint` routine (`PRINTBASE + 9`) after each page to flush the print buffer (if using a 7-bit printer then scanlines left in the buffer pointed to by r1 need to be printed out rather than combined with the next row of data) and to close the Commodore output file.

NOTE: `CARDSWIDE` is the number of Commodore hi-res bit-mapped cards wide.
`CARDSDEEP` is the number of Commodore hi-res bit-mapped cards deep.

For ASCII Printing:

(A) Call `InitForPrint (PRINTBASE)` once per page to initialize the printer.

(B) Call `SetNLQ (PRINTBASE+21)` if printing in near letter quality mode is desired.

(C) Call `StartASCII (PRINTBASE + 18)` once per page to set up the Commodore

file to output on the serial bus. Any errors are returned in x and the carry bit is set. If no error was detected, x is returned with 0.

(D) To print out each row of cards (do 1,2, and 3 for each line) do the following. (1) Load a buffer with a string of ASCII character data and load r0 with the start address of the buffer. Append a CR to the end of each line to cause a CR and LF to be output by the printer. (2) Load r1 with the start address of 640 bytes RAM for the print routines to use. (3) Call the PrintASCII routine (PRINTBASE + 15). NOTE: Unlike PrintBuffer, r1 need not point to the same memory for the whole document, or be preserved between calls to PrintBuffer. r0 can change each time PrintBuffer is called. Goto 1 until document is complete.

(E) Call the StopPrint routine (PRINTBASE + 9) at the end of every page to form feed to the next page. and to close the Commodore output file.

We now describe these routines in greater detail. After this section we present two sample printer drivers. The first is for Commodore compatible printers. This driver is a good model for any 60 dot per inch printer. Following the Commodore driver is the driver for the Epson FX series of printers. This driver is a good model for any 80 dot per inch printer.

InitForPrint

Function: Perform printer dependent initialization once per document.

Pass: nothing.

Return: nothing.

Destroyed: Depends on printer.

Synopsis: InitForPrint is the first of the routines to be called. It sends the printer any special commands it needs to be in to print GEOS documents. In some printer drivers this routine does nothing.

GetDimensions

Function: Return the dimensions in cards of the rectangle that will print in an 8 x 10.5 inch area of the screen.

Pass: nothing

Return: x - width in cards
y - height, in card rows which will print

Destroyed: nothing

Synopsis: **GetDimensions** returns to the application the resolution of the page the printer is capable of printing in an 8 x 10.5 inch area. Some low-res printers can only print 60 dots per inch and thus cannot print the entire 640 pixel width of GEOS document. The application must decide what portion of the page to display, and what to sacrifice. **geoPaint** will print the leftmost 60 cards. **geoWrite** clips a small portion, 10 cards, from the left of an embedded graphic when it fills the 640 byte print buffer with data and lets the printer driver print the next 60 cards, thus printing the middle 60 of the 80 card width. (The printer driver won't print the last 10 cards on the line.)

Upon return, **GetDimensions** will place the width of the line in x. It loads y with the number of times to fill up the buffer, i.e. the number of rows (made up of 8 scanlines each) which make up a page. **GetDimensions** loads the a, x, and y registers from a table located at the start of the printer dependent file and returns.

It is unnecessary to call **GetDimensions** when printing in draft or NLQ modes. All ascii printing is done on a 66 lines per page, 80 characters per line basis.

StartPrint

Function: Initializes the serial bus to talk to the printer. Sets the printer to receive graphics data.

Pass: nothing

Return: a, x, y, r3

Destroyed: Depends on the printer

Synopsis: **StartPrint** is called to open up a "fake" Commodore file structure for sending characters to the printer on the serial bus. A logical file to the printer is not actually opened, the purpose is so that the printer thinks a file is open, pointing to it. This is so that Kernal ROM routines may be used to talk to the printer. The printer is device number 4. The routine calls SetDevice to tell GEOS that the printer is being addressed, and InitForIO to set the memory map to include the Kernal ROM and the i/o space and to disable interrupts. It then uses the Kernal ROM primitives to open a file to the printer. Any errors encountered in the Kernal routines are returned are in location \$90, and if \$90 is non-zero, it is loaded into the x register, the file is closed, and the routine returns to the application. The value returned in the x register is the same number returned by the Kernal ROM routines. The two errors "device not present" and "I/O timeout errors" is done should be checked by the application here. If the file is successfully opened, the printer is told to listen, and any printer dependent parameters are set up. The printer is then told to un-listen and after a short delay. DoneWithIO is called to return the memory map to the application's state. On return the x register is loaded with zero to tell the application that no errors occurred. Remember, if there were no errors, on return from this routine, the printer thinks a Commodore output file is open pointing to the printer.

PrintBuffer

Function: Prints a 640 byte buffer (80 cards) of graphics data to the printer

Pass: r0 - address of the 640 bytes (80 cards) to be printed.
r1 - address of an additional 640 byte buffer for PrintBuffer to use.

Note: this buffer may not change between calls to PrintBuffer. 7 bit printers use it to store the left over scanlines between calls. Each time PrintBuffer is called it is passed 8 scanlines of data but only 7 may be printed.

Return: r0, r1 - unchanged

Destroyed: a, x, y, r3

Synopsis: **PrintBuffer** is called to print one line of hi-res bit-mapped cards from the buffer pointed at by r0 (the user buffer). It again does a SetDevice and an InitForIO before telling the printer to listen. If the printhead can print out 8 bits, PrintBuffer simply prints the user buffer pointed to by r0.

If the printhead can only print 7 bits, then more work has to be done since the last scanline of data won't be printed. The first call to PrintBuffer will cause the first 7 scanlines to be printed. The leftover line is then moved to an internal buffer. The next time PrintBuffer is called by the application, 6 scanlines of data from the new buffer are added to the leftover scanline and printed. There are now two left over scanlines. Printing continues this way. Every 56 (8*7) scanlines printed, PrintBuffer will actually print two 7 pixel high rows.

After printing one line of graphic data, the PrintBuffer routine sends a carriage return-linefeed if needed (determined by a routine in the printer dependent file), tells the printer to unlisten, and calls DoneWithIO to return the memory map to the application's state. For more information see the section on sample printer drivers.

StopPrint

Function: Called at end of every page to flush output buffer and tell the printer to form feed.

Pass: r0 - address of the 640 bytes (80 cards) to be printed.
r1 - address of an additional 640 byte buffer for PrintBuffer to use.
Note: this buffer may not change between calls to PrintBuffer. 7 bit printers use it to store the left over scanlines between calls. Each time PrintBuffer is called it is passed 8 scanlines of data but only 7 may be printed.

Return: r0, r1 - unchanged

Destroyed: a, x, y, r3, r2

Synopsis: **StopPrint** is called after all cards for a given page have been sent to the printer. It does a SetDevice, InitForIO, makes printer listen, and if the printhead was printing 7-bit high data, flushes out any remaining lines of data in the print buffer. It then does a form-feed and an unlisten, closes the "fake" Commodore output file, and does a DoneWithIO.

StartASCII

Function: Initializes the serial bus to talk to the printer. Sets the printer to receive streams of ASCII data.

Pass: nothing

Return: a, x, y, r3

Destroyed: Depends on the printer

Synopsis: **StartASCII** does much the same as **StartPrint**. A fake file to the printer is opened so that the printer thinks a file is open, pointing to it. This is so that Kernal ROM routines may be used to talk to the printer. Instead of graphics data mode the printer is initialized for receiving ASCII data. See **StartPrint** for more details.

PrintASCII

Function: Prints the null terminated ASCII string pointed to by r0.

Pass: r0 - pointer to the ASCII string.
r1 - address of an additional 640 byte buffer for PrintBuffer to use.

Return: nothing

Destroyed: a, x, y, r3

Synopsis: **PrintBuffer** is called to print one line of ASCII data in the printers resident character set from the buffer pointed at by r0 (the user buffer). It does a **SetDevice** and an **InitForIO** before telling the printer to listen. If a carriage return is encountered in the string, both it and a line feed are sent to the printer.

After printing one line of graphic data, the **PrintASCII** routine tells the printer to unlisten, and calls **DoneWithIO** to return the memory map to the application's state.

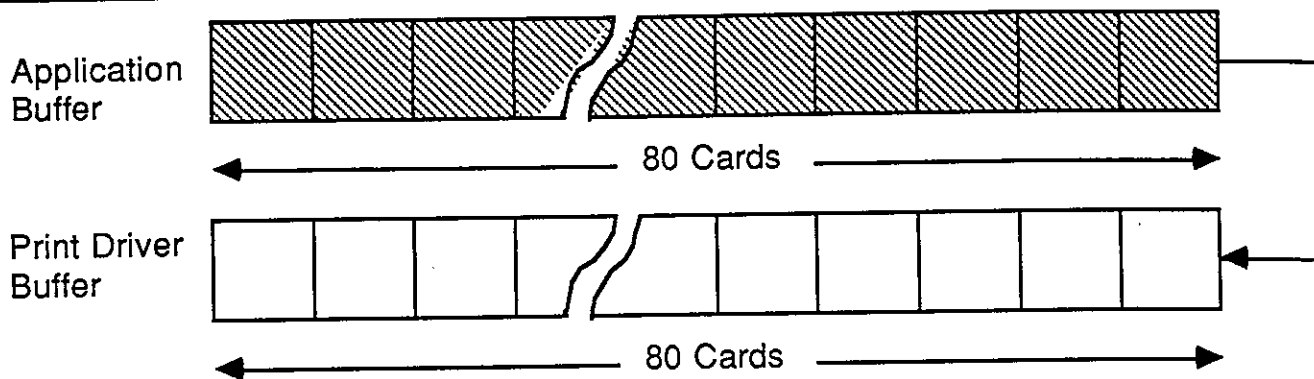
We now present two sample printer drivers, one for a 7-bit printer and the other for an 8-bit one. These sample drivers ought to give the programmer a good idea of how to write a driver of his own.

SamplePrinterDriver

Introduction to Sample Driver

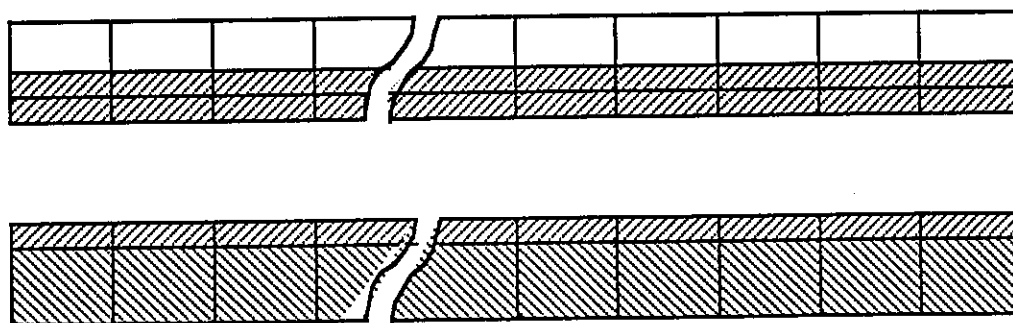
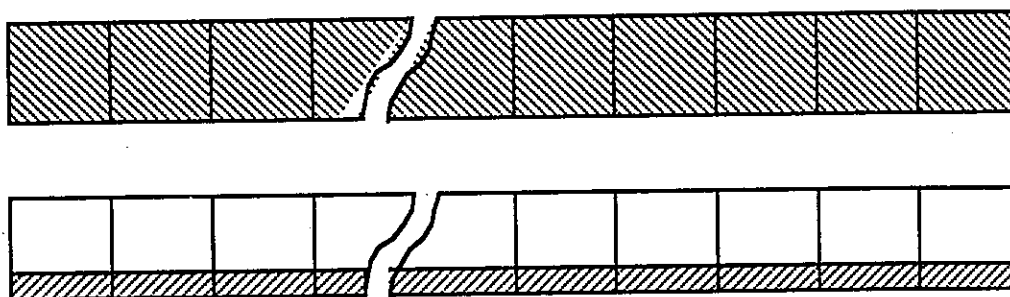
Two basic printer drivers provide the prototypes for the remainder of drivers in existence, one for 7-bit and one for 8-bit printers. These two types of drivers differ in that the 7-bit high printers can only print out 7 scanlines of data at one time. Since we pass 8-bit data to the printers, one scanline of data must be saved after the first call to PrintBuffer and joined with the next set of data. The second time PrintBuffer is called it prints the leftover scanline along with six scanlines from the eight just passed. Two scanlines will be left over. By the time 56 scanlines have been passed, PrintBuffer will have enough left over to print two 7 scanlines high rows. It will have six left over, print them with one from the newly passed eight and then print the seven left over.

The diagram below shows the first few step in the printing out of a page.



Application Passes Data in 640 Byte Buffer
Data in Application Print Buffer is transferred to Print Driver Buffer.

After printing, PrintBuffer returns and the Application reads in new buffer of data. The Printer Driver buffer holds the left over scanline.



Data Shifted to top of Printer Buffer. 6 lines of data from Application buffer are shifted in. 2 scanlines of data remain in Application buffer.

Printing with a 7-bit High Printer

The first panel shows the application has passed a full buffer to the printer driver. the printer driver then copies the data into its buffer for printing. In the second panel the printer driver has printed the top 7 scanlines of its buffer, sent a CRLF to the printer, and left one scanline unprinted. The application has also reloaded its buffer with 8 more scanlines of data. In the third panel, the leftover scanline in the printer driver's buffer has been shifted to the top and 6 scanlines of data have been shifted in from the application's buffer to fill up the lower part of the buffer. The PrintBuffer routine is now ready to start printing out the buffer.

It should be clear then that the printer driver needs its own 640 byte buffer to save scanlines between calls from the application so that it may combine the leftover lines with incoming lines.

The 8-bit printers avoid all this shifting around of data. They print the entire buffer of data at each call to PrintBuffer. Both types of drivers, however, must take some pains to "rotate" the data, which is to say assemble the horizontal bytes into vertical bytes for transmission over the serial bus. The first byte to be sent to the printhead is made up of the seventh bit from each of the first 8 (or 7 for a 7 bit printer) bytes in the first card. One bit at a time is shifted out from each of the bytes in the first card. Some printers put the bit from the first byte on top and others on the bottom.

We now turn to a sample printer driver for an 8-bit printer, the Epson FX80. Later we will present the algorithm we use to deal with 7-bit printers such as the Commodore 801.

8-Bit Printer Driver

For:

EPSON FX-80,FX-100,RX-80,RX-100,JX-80

PANASONIC KX-1091,KX-1092,KX-1592,KX-1595

tested on:

EPSON JX-80

.if(0)

PRINTER EQUATES

.endif

.include	gpquates	; Printer equates shown below
CARDSWIDE	= 80	;80 commodore cards wide.
CARDSDEEP	= 90	;90 commodore cards deep.
SECADD	= TRANSPARENT	;secondary address

.include	Macros6500	;see Appendix for these
.include	Constants	;Berkeley Assembler macros
.include	Memory_map	;GEOS Constants
.include	Routines	;Ram allocation and variables
		;jump table for GEOS routines

.psect PRINTBASE ; Execution address for printer driver

```
.if(0)
```

RESIDENT JUMP TABLE

```
.endif
```

```
InitForPrint:                ;first routine at PRINTBASE
    rts
    nop
    nop
StartPrint:                  ;second routine at PRINTBASE + 3
    jmp    r_StartPrint
PrintBuffer:                 ;third routine at PRINTBASE + 6
    jmp    r_PrintBuffer
StopPrint:                   ;fourth routine at PRINTBASE + 9
    jmp    r_StopPrint
GetDimensions:               ;fifth routine at PRINTBASE + 12
    jmp    r_GetDimensions
PrintASCII:                  ;sixth routine at PRINTBASE + 15
    jmp    r_PrintASCII
StartASCII:                  ;7th routine at PRINTBASE + 18
    jmp    r_StartASCII
SetNLQ:                      ;8th routine at PRINTBASE + 21
    jmp    r_SetNLQ
```

```
.if(0)
```

RAM STORAGE/ UTILITIES

```
.endif
```

```
PrinterName:                .byte    "Epson FX-80",0    ;name of printer as it should
                                                                ;appear in menu

prntblcard:                  .byte 0,0,0,0,0,0,0,0        ;printable character block
breakcount:                  .byte 0                      ;
reduction:
cardwidth                     .byte 0                      ;width of the eprint buffer line
                                                                ;in cards. Used for reduction
                                                                ;flag in laser drivers.
scount:                      .byte 0                      ;string output routine counter

cardcount:                   .byte 0
modelflag:                   .byte 0                      ;either 0=graphics, or $FF=ASCII
                                                                ; for draft or nlq mode
.include      utilities      ;utility routines, (see below).
```


Resident Top-Level Routines

GetDimensions

Function: Return the dimensions in cards of the rectangle that will print in an 8 x 10 area of the screen.

Pass: nothing

Return: X = width, in cards, that this printer can put out across a page
Y = height, in cards, that this printer can put down a page

Destroyed: nothing

Synopsis: GetDimensions returns the number of cards wide and high that this printer is capable of printing out on an 8 x 10 subset of an 8 1/2 by 11 inch page.

r_GetDimensions:

ldx	#CARDSWIDE	; get the number of cards wide
ldy	#CARDSDEEP	; and get the number of cards high
lda	#0	; set for graphics only driver.
rts		

```
.if(0)
```

StartPrint

Function: Performs initialization necessary before printing each page of a document.

Called By: A GEOS Application.

Pass: nothing

Return: nothing

Destroyed: a, x, y, r3

Synopsis: This is the StartPrint routine as discussed above. It initializes the serial bus to the printer, sets up the printer to receive graphic data.

```
.endif
```

```
r_StartPrint:
```

```
    lda    #0                ;set for graphic mode
    sta    modeflag          ;
```

```
StartIn:
```

```
    lda    #PRINTADDR        ;set to channel 4.
    jsr    SetDevice
    jsr    InitForIO          ;set I/O space in, ints dis.
    lda    #0
    sta    $90                ;init the error byte to no error.
    jsr    OpenFile           ;open the file for the printer.
    lda    $90                ;if problems with the output channel, go to
    bne    20$                ;error handling routine.
    jsr    OpenPrint          ;open channel to printer.
    jsr    InitPrinter        ;initialize the printer for graphic mode.
    jsr    ClosePrint         ;close the print channel.
    jsr    Delay              ;wait for weird timing problem.
    jsr    DoneWithIO         ;set mem map back, and enable ints.
    ldx    #0
    rts
```

```
20$:
```

```
    pha                ;save error return from the routines.
    ;    bit 0 set: timeout, write.
    ;    bit 7 set: device not present.
```

```
jsr    CloseFile    ;close the file anyway.
jsr    DoneWithIO   ;set mem map back, and enable ints.
pla     ;recover the error return.
tax     ;pass out in x.
rts
```

Delay:

```
ldx    #0
10$:   ldy    #0
20$:   dey
       bne    20$
       dex
       bne    10$
       rts
```

.if (0)

r_StartASCII

Function: Initializes the Epson to receive ASCII print streams

Called By: A GEOS Application.

Pass: nothing

Return: nothing

Destroyed: a

Synopsis: Just sets the mode flag. called by user at beginning of each document

.endif

r_StartASCII:

LoadB	modeflag,\$FF	;set mode to ascii printing
jmp	StartIn	;pick up rest of start Print.

```
.if(0)
```

r_SetNLQ

Function: Initializes the Epson to near letter quality mode

Called By: A GEOS Application.

Pass: nothing

Return: nothing

Destroyed: a

Synopsis: Send the printer driver the correct initialization bytes to put it in nlq mode.

```
.endif
```

r_SetNLQ:

<code>lda</code>	<code>#PRINTADDR</code>	<code>;set to channel 4.</code>
<code>jsr</code>	<code>SetDevice</code>	<code>;set device number</code>
<code>jsr</code>	<code>InitForIO</code>	<code>;put io space in, disable interrupts</code>
<code>jsr</code>	<code>OpenPrint</code>	<code>;open channel to printer.</code>
<code>LoadW</code>	<code>r3,#nlqtbl</code>	<code>;table of initialization bytes to send</code>
<code>lda</code>	<code> #(enlqtbl-nlqtbl)</code>	<code>;the length of the talbe</code>
<code>jsr</code>	<code>Strout</code>	<code>;send the table to the printer</code>
<code>jsr</code>	<code>ClosePrint</code>	<code>;close the print channel.</code>
<code>jsr</code>	<code>DoneWithIO</code>	<code>;put RAM back in, enable interrupts</code>
<code>rts</code>		

.if(0)

PrintASCII

Function: Prints a null terminated ASCII string to the printer

Called By: A GEOS Application.

Pass: r0 = pointer to the ASCII string
r1 = pointer to the 640 bytes buffer for the printer to use.

Return: nothing

Destroyed: assume all registers

Synopsis: Sends a null terminated ASCII string to the printer. All carriage returns and linefeeds must be handled by the application. Carriage returns are mapped into cr-lf.

.endif

PrintASCII:

```

        lda #PRINTADDR           ;set to channel 4.
        jsr SetDevice            ;set to printer device
        jsr InitForIO            ;put i/o space in and disable interrupts
10$:
        ldy #0                   ;init the index into ascii string
        lda (r0),y               ;get the character
        beq 30$                  ;if at end of string, exit.
        cmp #CR                  ;if carriage return, add LDF
        bne 20$                  ;branch if not CR
        jsr Ciout                 ;output the character
        lda #LF                  ;load up the cr
20$:
        jsr Ciout
        incw r0                  ;point to next character
        jmp 10$                  ;do again.
30$:
        jsr ClosePrint           ;close the print channel
        jsr DoneWithIO           ;put ram back in, enable interrupts
        rts

```

PrintBuffer

Function: Prints out the indicated 640 byte buffer of graphics data (80 cards) as created by an application.

Called By: A GEOS Application.

Pass: r0 - address of the 640 bytes (80 cards) to be printed.
r1 - address of an additional 640 byte buffer for PrintBuffer to use.

Note: this buffer may not change between calls to PrintBuffer. 7 bit printers use it to store the left over scanlines between calls. Each time PrintBuffer is called it is passed 8 scanlines of data but only 7 may be printed.

Return: r0, r1 - unchanged

Destroyed: a, x, y, r3

Synopsis: PrintBuffer, as described in more detail above, is the top level routine that dumps data from the GEOS 640 byte buffer maintained in the Commodore c64 to the printer using the serial port.

r_PrintBuffer:

lda	#PRINTADDR	;set to channel 4.
jsr	SetDevice	;set GEOS device number.
jsr	InitForIO	;put IO space in, dis ints.
jsr	OpenPrint	;open channel to printer.
MoveW	r0,r3	
jsr	PrintPrintBuffer	;print the users 8 bit high buffer.
jsr	Greturn	;do cr-lf here.
jsr	ClosePrint	;close the print channel.
jsr	DoneWithIO	;put back the mem map, enabl ints.
rts		;exit.

StopPrint

Function: Called at end of every page to flush output buffer and tell the printer to form feed.

Pass: r0 - address of the 640 bytes (80 cards) to be printed.
r1 - address of an additional 640 byte buffer for PrintBuffer to use.
Note: this buffer may not change between calls to PrintBuffer. 7 bit printers use it to store the left over scanlines between calls. Each time PrintBuffer is called it is passed 8 scanlines of data but only 7 may be printed.

Return: r0, r1 - unchanged

Destroyed: a, x, y, r3

Synopsis: StopPrint is called after all cards for a given page have been sent to the printer. It does a SetDevice, InitForIO, makes the printer listen, and if the printhead was printing 7-bit high data, flushes out any remaining lines of data in the print buffer. It then does a form-feed and an unlisten, closes the Commodore output file, and does a DoneWithIO.

r_StopPrint:

lda	#PRINTADDR	;set to channel 4.
jsr	SetDevice	;set geos device number.
jsr	InitForIO	;put io space in, dis ints.
jsr	OpenPrint	;open channel to printer.
jsr	FormFeed	;do a form feed
jsr	ClosePrint	;close the print channel.
jsr	CloseFile	;close the print file.
jsr	DoneWithIO	;put back the mem map, enabl ints.
rts		

.if(0)

Resident Subroutines

PrintPrintBuffer

Function: Prints out the print buffer pointed to by r3**Called by:** PrintBuffer**Pass:** r3 - address of start of buffer to print**Return:** r3 - unchanged**Destroyed:** a, x, y, r0 - r15**Synopsis:** Checks to see if the buffer is empty before printing the data. Then for each card in the buffer, rotate the data and send it to the printer.**PrintPrintBuffer:**

```

        PushW    r3                ;save the buffer pointer.
        jsr      TestBuffer        ;see if the buffer is all zeros.
        bcs      5$                ;if there is data in the buffer, send it.
        PopW     r3                ;dummy pop.
        rts

5$:
        jsr      SetGraphics        ;set graphics mode for this line.
        PopW     r3                ;restore the buffer pointer.
        lda      #CARDSWIDE        ;load the card count ( up to 80).
        sec
        sbc      cardcount
        tax

10$:
        txa                      ;save x.
        pha
        jsr      Rotate            ;rotate the card....
        jsr      SendBuff          ;send the rotated card.
        AddVW    #8,r3            ;update pointer to buffer.
        pla                      ;recover x.
        tax
        dex
        bne      10$              ;done?
        rts                      ;if not, do another card.

```

TestBuffer

Function: Tests buffer to see if there is anything to print.

Called by: PrintPrintBuffer

Pass: r3 - pointer to beginning of print buffer to test

Return: carry - carry flag = 1 if data in the buffer, else carryflag = 0

Destroyed: a, x, r3

Synopsis: Check all the bytes in the buffer to see if all are \$00.

.endif

TestBuffer:

```

LoadB    cardcount, #0
ldx      #7                ;assume 8 bit high printhead.
stx      scout            ;save.
AddVW    #((CARDSWIDE-1)<<3), r3
ldx      #CARDSWIDE        ;load the cards / line.
10$:
ldy      scout            ;get the line count.
20$:
lda      (r3), y           ;check a byte.
bne      30$              ;if zero , skip to check another byte.
dey      ;point at next byte in card.
bpl      20$              ;if not at end, check next byte in this card.
SubVW    #8, r3            ;point at next card.
inc      cardcount
dex      ;see if all the cards are done.
bne      10$              ;if not done, do another card.
clc      ;if here, then line was clear.
rts
30$:
sec      ;set the carry to signal data was found
rts

```

InitPrinter

Function: Initializes the Epson to line-feed 8/72".

Called by: StartPrint

Pass: nothing

Return: r3 - #inittbl
scount - \$FF
y - 0

Destroyed: a

Synopsis: Outputs to the printer a string of characters which initializes it. See the printer's owners manual.

.endif

InitPrinter:

```

    bit    modeflag      ;see if printing ascii or graphic mode
    bmi    10$           ;branch if ascii
    LoadW r3,#inittbl   ;table of bytes for initialization.
    lda    #(einittbl-inittbl) ;length of string.
    jmp    Strout        ;output the string.
10$:
    LoadW r3,#ainittbl   ;table of bytes for ascii initialization.
    lda    #(eainittbl-ainittbl) ;length of string.
    jmp    Strout        ;output the string.
inittbl:
    .byte  8,"A",ESC     ;send 8/72" line feed
    .byte  "@",ESC       ;reset totally
einittbl:
ainittbl:
    .byte  2,ESC         ;send 6 lines/inch
    .byte  "@",ESC       ;reset totally
aeinittbl:
```

SetGraphics

Function: Sets the epon into 640 column graphics mode.

Called by: PrintPrintBuffer

Pass: cardcount - the number of the card being processed.

Return: r3 - #wsdgphtbl, the printer width table
 scount - \$FF
 y - 0

Destroyed: a

Synopsis: Tell printer the graphics mode and how many bytes to expect.

SetGraphics:

```

LoadB    r3h,#0                ;clear top byte.
MoveB    cardcount,r3l         ;load cardcount into low byte.
asl      r3l                   ;x 8.
rol      r3h
asl      r3l
rol      r3h
asl      r3l
rol      r3h
sec
lda      #[(CARDSWIDE<<3)      ;get total width for the page(bytes).
sbc      r3l
sta      wsdgphtbl+1
lda      #](CARDSWIDE<<3)      ;get total width for the page(bytes).
sbc      r3h
sta      wsdgphtbl
LoadW    r3,#wsdgphtbl         ;table of control bytes for 640 col
                                           ;sgl den.
lda      #(ewsdgphtbl-wsdgphtbl) ;length of string.
jmp      Strout                ;output the string.

```

wsdgphtbl:

```

;ESC * 4: set screen dump mode (80 dpi in graphics mode).
;N1 N2: set the number of graphic bytes to output.
.byte    2,128,4,"*",ESC

```

ewsdgphtbl:

Function: Sends a printable card out the serial port.

Pass: prntblcard - this buffer contains the rotated card data

Destroyed: a, x

Synopsis: After a card has been rotated so that the bytes each represent a vertical column of bits to go to the printer, SendBuff sends the card accross the serial bus.

```

SendBuff:
    ldx        #0                ;initialize the count.
10$:
    txa
    pha
    lda        prntblcard,x      ;get byte to send.
    jsr        Ciout             ;send this byte
    pla
    tax
    inx
    cpx        #8                ;are we done with all bytes?
    bne        10$              ;if not, continue with sending.
    rts

```

Greturn FormFeed

Function: Prints out a graphic linfeed and carriage return.
Exit graphics mode and print the form feed char.

Called by: PrintBuffer

Pass: nothing

Return: nothing

Destroyed: a, vars destroyed by Ciout

Synopsis: Outputs the CRLF

Greturn:

```
lda    #CR           ;carriage return.
jsr    Ciout         ;send it.
lda    #LF           ;line feed.
jsr    Ciout         ;send it.
rts
```

FormFeed:

```
lda    #FF
jsr    Ciout
rts
```

Rotate

Function: Rotates a hi res bit mapped card from the 640 byte print buffer to an 8 byte buffer which is then ready for sending to the printer.

Called by: PrintPrintBuffer

Pass: r3 - address of the card to be operated on.

Return: prntblcard - rotated data placed here

Destroyed: a, x, y

Synopsis: Create the nth byte in the prntblcard buffer out of the nth bit of each of the bytes in the card pointed to by r3. This rotates a hires bit mapped card from the 640 byte print buffer pointed at by r3 into the prntblcard 8 byte buffer.

Rotate:

```

sei                ;disable any IRQs.
ldy                ;initialize the index into the card.
#7
10$:
lda                ;get the byte from the card.
(r3),y
ldx                ;initialize the index into the printable card.
#7
20$:
ror                ;get the least significant bit into c.
a                  ;shift it into the printable card table.
ror                ;next bit.
prntblcard,x
dex                ;if not done, store another bit.
bpl                ;next byte.
20$
dey                ;if not done, load another byte.
bpl                ;clear interrupt disable bit.
10$
cli
rts

```

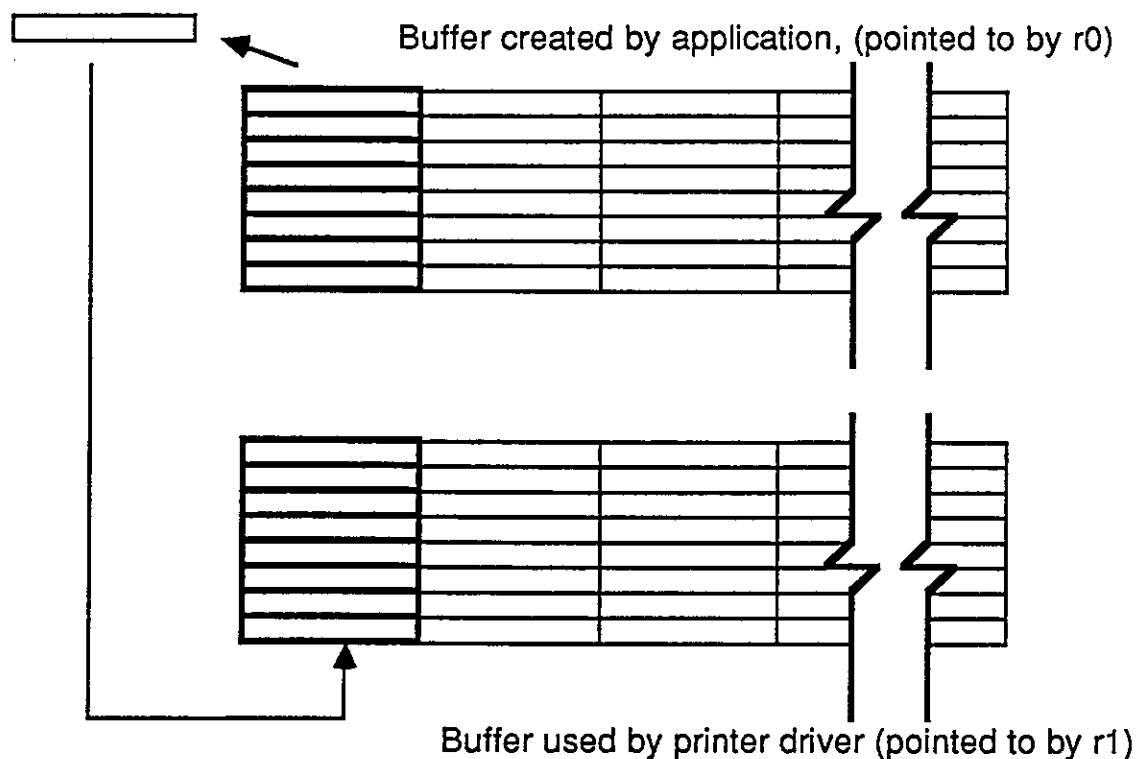
PRINTEND: ;last label in Epson FX Printer driver



19.

Commodore Driver

The Commodore driver is similar in overall structure to the Epson driver presented earlier. The fact that the Commodore printer is a 7-bit printer makes life a bit harder. The 8-byte high card oriented buffer must itself be buffered into so that it may be printed 7 scanlines at a time. This is done in routines TopRollBuffer and BotRollBuffer. TopRollBuffer calls RollaCard to take a byte off the top of a card in the print buffer and shift each byte in the card up one as shown below.



Roll a byte from the user buffer to the internal buffer.

After the line is printed, there will be left over lines in the user buffer that will be printed the next time PrintBuffer is called. (Remember that with 7-bit printers, PrintBuffer can only print 7 of the 8 scanlines passed from the application in the buffer pointed to by r0. This leaves one scanline of data left over after the first call to PrintBuffer.) BotRollBuffer rolls these leftover lines into the internal print buffer. For example, before the first line is printed, TopRollBuffer rolls the top 7 lines from the user print buffer to the internal printer driver buffer. These lines are printed and then BotRollBuffer is called to shift the remaining scanline from the user buffer to the internal buffer. PrintBuffer then returns to the application which is now free to reload the user buffer. TopRollBuffer and BotRollBuffer read a table to determine how many scanlines to roll each time they are called. The actual rolling of the scanlines is done a card at a time because the bytes in the user print buffer are organized that way. It was decided to have the application pass its output graphics data in card format since it is probable that most of the routines for drawing to the screen could then be reused to create the data for the printer.

Included below is an assembler listing of the driver for Commodore compatible printers.

Commodore Compatible Printer Driver Description

For:

COMMODORE MPS-801,MPS-803,MPS-1000,1525
 ERGO SYSTEMS HUSH 80CD
 OKIDATA OKIMATE 10
 SEIKOSHA SP-1000VC

tested on:

COMMODORE MPS-801,MPS-1000
 ERGO SYSTEMS HUSH 80CD
 SEIKOSHA SP-1000VC

to print out a document, the calling application must :

.if(0)

PRINTER EQUATES

.endif

.include	gpquates	; Printer equates shown below
CARDSWIDE	= 60	;80 Commodore cards wide.
CARDSDEEP	= 94	;90 Commodore cards deep.
SECADD	= LOWERCASE	;secondary address

;see Appendix for these

.include	Macros6500	;Berkeley Assembler macros
.include	Constants	;GEOS Constants
.include	Memory_map	;RAM allocation and variables
.include	Routines	;jump table for GEOS routines
.psect	PRINTBASE	; Execution address for printer ; driver

```
.if(0)
```

.if(0)

PRINT DRIVER TO DISK STUFF

.endif

;include Berkeley Softworks' own macros and constants

.include Macros6500
.include Constants
.include Memory_map
.include Routines

PRINTBASE = \$7900 ; location of driver in memory,
; \$7900 is tail end of Background Screen.
; The driver may occupy through \$7F3F.
; (just under 2K)
.psect PRINTBASE ; Execution address for final printer driver

.endif

Resident Top-Level Routines

.if(0)

StartPrint

Synopsis: StartPrint initializes the serial bus to the printer, sets up the printer to receive graphic data, and initialize the break count RAM location.

Called by: GEOS Application.

Pass: nothing

Destroyed: a, x, y, r3

.endif

r_StartPrint:

LoadB modeflag, #0

StartIn:

```

lda    #PRINTADDR    ;set to channel 4.
jsr    SetDevice
jsr    InitForIO      ;set I/O space in, ints dis.
lda    #0              ;initialize the counter for the card breaks.
sta    breakcount
sta    $90             ;init the error byte.
jsr    OpenFile        ;open the file for the printer.
lda    $90             ;if problems with the output channel, go to
bne    20$             ;error handling routine.
jsr    Delay           ;wait for weird timing problem.
jsr    DoneWithIO      ;set mem map back, and enable ints.
ldx    #0
rts

```

20\$:

```

pha                ;save error return from the routines.
                  ;a might have bit 0 set:
                  ;    timeout, write.
                  ;    bit 7 set:
                  ;    device not present.
jsr    CloseFile     ;close the file anyway.
jsr    DoneWithIO     ;set mem map back, and enable ints.
pla                ;recover the error return.
tax                ;pass out in x.
rts

```

```
Delay:      ldx      #0
10$:        ldy      #0
20$:        dey
            bne      20$
            dex
            bne      10$
            rts
```

```
.page
```

```
.if (0)
```

PrintBuffer

Synopsis: PrintBuffer is the top level routine that dumps data from the GEOS 640 byte buffer maintained in the c64 to the printer using the serial port .

Called by: GEOS Application.

Pass: r0 = address of the 640 bytes (80 cards) to be printed.
 r1 = address of an additional 640 bytes of data memory
 to be used by the PrinttoDevice routine.

IMPORTANT.....

this memory pointed at by r1 MUST stay intact between calls to the PrinttoDevice routine. It is used as a storage area for the partial lines for the 7 bit high printers.

Aaccessed: nothing

Returned: r0, r1 preserved

Ruturned (probably not useful but consistant): nothing

Destroyed: a, x, y, r3

```
.endif
```

r_PrintBuffer:

```
lda    #PRINTADDR           ;set to channel 4.
jsr    SetDevice             ;set GEOS device number.
jsr    InitForIO             ;put I/O space in, dis ints.
jsr    OpenPrint             ;open channel to printer.
jsr    i_PrintBuffer         ;print out a line.
jsr    ClosePrint            ;close the print channel.
jsr    DoneWithIO            ;put back the mem map, enabl ints.
rts                                     ;exit.
```

i_PrintBuffer:

```
jsr    TopRollBuffer         ;roll into print buffer.
MoveW   r1,r3
jsr    PrintPrintBuffer       ;print it .
jsr    BotRollBuffer          ;roll leftover lines into print buffer.
lda     breakcount            ;see if we just print the last line in brktab
cmp     #7
bne     5$                   ;if not, skip.
MoveW   r1,r3                 ;point to print buffer.
jsr    PrintPrintBuffer       ;print it again.
lda     #0                    ;stuff breakcount.
sta     breakcount
```

5\$:

```
inc     breakcount            ;next index to breaks in 7 bit printing.
                                   ;valid values = 1-7
rts
```

.page

.if (0)

StopPrint

Synopsis: StopPrint is called when a page of a document is finished or when the document itself is finished.

Called by: GEOS Application.

Pass: r0 = address of the 640 bytes (80 cards) to be printed.
r1 = address of an additional 640 bytes of data memory to be used by the PrintToDevice routine.

IMPORTANT.....

this memory pointed at by r1 **MUST** stay intact between calls to the PrintToDevice routine. It is used as a storage area for the partial lines for the 7 bit high printers.

Accessed: r3

Destroyed: a, x, y, r3

.endif

r_StopPrint:

```

    lda    #PRINTADDR          ;set to channel 4.
    jsr    SetDevice            ;set GEOS device number.
    jsr    InitForIO            ;put io space in, dis ints.
    jsr    OpenPrint            ;open channel to printer.
    bit    modeflag             ;if ASCII printing...
    bmi    10$                  ;skip buffer flush.
    PushW   r0                  ;save the buffer addresses.
    PushW   r1
    MoveW   r0,r1                ;load the address of RAM to clear.
    LoadW  r0,#640              ;length to clear.
    jsr    ClearRam             ;clear it.
    PopW    r1                  ;recover the buffer addresses.
    PopW    r0
    jsr    i_PrintBuffer        ;flush out the buffer data.
10$:
    jsr    FormFeed              ;do a form feed
    jsr    ClosePrint           ;close the print channel.
    jsr    CloseFile            ;close the print file.
    jsr    DoneWithIO           ;put back the mem map, enabl ints.
    rts

```

.page

.if (0)

GetDimensions

Synopsis: GetDimensions returns the number of cards wide and high that this printer is capable of printing out on an 8 1/2 by 11 inch page.

Called by: GEOS Application.

Pass: nothing

Accessed: nothing

Returned: X = width, in cards, that this printer can put out across a page
Y = height, in cards, that this printer can put down a page

Destroyed: nothing

.endif

r_GetDimensions:

ldx	#CARDSWIDE	; get the number of cards wide
ldy	#CARSDDEEP	; and get the number of cards high
lda	#0	;set for graphics only driver.
rts		

.if (0)

r_StartASCII

Synopsis: Sets the Commodore up to receive ASCII print streams.

Called by: GEOS application

Pass: nothing

Accessed:

Returned:

Returned

Destroyed:

.endif

r_StartASCII:

LoadB modeflag,\$ff ;set mode to ASCII printing.
jmp StartIn

.if (0)

PrintASCII

Synopsis: Prints a null terminated ASCII string passed in the buffer pointed at by r0.

Called by: GEOS Application.

Pass: r0 = pointer to ASCII string
r1 = pointer to 640 bytes for the print driver to use.

Accessed:

Returned:

Returned

Destroyed:

.endif

```
r_PrintASCII:
    lda    #PRINTADDR          ;set to channel 4.
    jsr    SetDevice           ;set GEOS device number.
    jsr    InitForIO           ;put I/O space in, dis ints.
    jsr    OpenPrint           ;open channel to printer.

10$:
    ldy    #0                  ;init the index.
    lda    (r0),y              ;get the character.
    beq    30$                 ;if null terminator, exit.
    cmp    #65                  ;see if alpha char, for CBM ASCII conversion.
    bcc    12$                 ;if < 'A', skip
    cmp    #123
    bcs    12$
    eor    #$20                ;convert upper to lower and vice-versa

12$:
    jsr    Clout
    IncW    r0
    jmp    10$                 ;do again.

30$:
    jsr    ClosePrint          ;close the print channel.
    jsr    DoneWithIO          ;put back the mem map, enabl ints.
    rts
```

Resident Subroutines

.if(0)

PrintPrintBuffer

Synopsis: Prints out the Print buffer pointed at by r3.

Called by: PrintBuffer

Pass: r3 = start of buffer to print.

Returned (probably not useful but consistant):

Destroyed:

.endif

PrintPrintBuffer:

	PushW	r3	;save the bufffer pointer.
	jsr	TestBuffer	;see if the buffer is all zeros.
	bcs	5\$;if there is data in the buffer, send it.
	PopW	r3	;dummy pop.
	jsr	SetGraphics	;get into graphics mode.
	jmp	20\$	
5\$:	jsr	SetGraphics	;set graphics mode for this line.
	PopW	r3	;restore the buffer pointer.
	lda	#CARDSWIDE	;load the card count (up to 80).
	sec		;adjust for the blank cards.
	sbc	cardcount	
	tax		;load x.
10\$:	txa		;save x.
	pha		
	jsr	Rotate	;rotate the card....
	jsr	SendBuff	;send the rotated card.
	AddVW	#8,r3	;update pointer to buffer.
	pla		;recover x.
	tax		
	dex		;done?
	bne	10\$;if not, do another card.

```

20$:
    jsr    Greturn      ;do graphics return here.
    jsr    UnSetGraphics ;get out of graphics mode.
    rts

```

```

.if (0)

```

TopRollBuffer

Synopsis: Rolls the entire print buffer up the correct amount of lines for the previously unprinted lines to be printed over any new lines in the user buffer.

Called by: PrintBuffer

Pass:

r0	=	pointer to user buffer.
r1	=	pointer to my print buffer.

Accessed:

Returned:

Returned (probably not useful but consistant):

Destroyed: a, x

```

.endif

```

TopRollBuffer:

```

    PushW   r0          ;save buffer pointers.
    PushW   r1
    ldx     #CARDSWIDE-1 ;load the card count .
10$:
    ldy     breakcount   ;get the count for the break table index.
    lda     topbreaktab,y ;get the number of lines to roll.
    jsr     RollaCard    ;rotate the card....
    dex
    ;done?
    bpl     10$          ;if not, do another card.
    PopW    r1           ;recover the pointers.
    PopW    r0
    rts
topbreaktab:
    .byte   8,7,6,5,4,3,2,1

```

.if (0)

BotRollBuffer

Synopsis: Rolls the entire print buffer up the correct amount of lines for the unprinted lines from the user buffer to be rolled into the bottom of the print buffer.

Called by: PrintBuffer

Pass: r0 = pointer to user buffer.
r1 = pointer to my print buffer.

Accessed:

Returned (probably not useful but consistant):

Destroyed: a, x

.endif

BotRollBuffer:

```

    PushW    r0            ;save buffer pointers.
    PushW    r1
    ldx      #CARDSWIDE-1  ;load the card count.
10$:
    lda      breakcount    ;get the count for the numbr of lines to roll.
    jsr      RollaCard     ;rotate the card....
    dex
    ;done?
    bpl      10$           ;if not, do another card.
    PopW     r1            ;recover the pointers.
    PopW     r0
    rts

```

.page

.if (0)

RollaCard

Synopsis: Rolls a card from the user buffer into the print buffer a lines.

Called by: TopRollBuffer,BotRollBuffer

Pass: a = number of lines to roll.

Accessed: r3l

Returned: r0 = r0+8, r1 = r1+8

Returned (probably not useful but consistant): r3l = 0

Destroyed: a

.endif

RollaCard:

```
    sta    r3l           ;store the loop count.
10$:    jsr    Roll8bytesOut ;shift out of the user buffer....
        jsr    Roll8bytesIn ;and into the print buffer.
        dec    r3l       ;done?
        bne    10$       ;if not, do another byte.
        AddVW  #8,r0      ;update pointer to user buffer.
        AddVW  #8,r1      ;update pointer to print buffer.
        rts
```

.page

.if (0)

TestBuffer

Synopsis: Tests if there is any thing on the current print line.**Called by:** PrintPrintBuffer**Pass:** r3 = pointer to the beginning of the current buffer.**Returned:** If any thing on line, c = 1 if not c = 0**Destroyed:** a, r3

.endif

TestBuffer:

```

    lda    #0                ;init the card count.
    sta    cardcount
    AddVW  #((CARDSWIDE-1)<<3),r3
    lda    #6                ;assume 7-bit high printhead (check the top 7).
    sta    scount            ;save.
    ldx    #CARDSWIDE-1      ;load the cards / line.
10$:      ldy    scount        ;get the line count.
20$:      lda    (r3),y        ;check a byte.
          bne    30$          ;if zero, skip to check another byte.
          dey    ;point at next byte in card.
          bpl    20$          ;if not at end, check next byte in this card.
          SubVW  #8,r3        ;point at next card.
          inc    cardcount     ;up date the number of blank cards.
          dex    ;see if all the cards are done.
          bpl    10$          ;if not done, do another card.
          clc    ;if here, then line was clear.
          rts
30$:      sec                ;set the carry to signal data was found
          rts

```

.if (0)

Roll8bytesIn

Synopsis: Rotates 8 bytes through a, used in the routines to load the second 640 byte print buffer. the effect is to roll a line of cards up 1 line.

Called by: -

Pass: a = byte to fill in at bottom of card.
 r1 = pointer to card to roll up 1 line.

Accessed:

Returned:

Returned (probably not useful but consistant):

Destroyed: a,y

.endif

Roll8bytesIn:

```

    pha                ;save the byte to fill in with.
    ldy                ;init the index to the card.
10$:
    iny                ;point at next line down (top byte is lost).
    lda                ;load a line from the card.
    dey                ;point at next line up.
    sta                ;store the byte at the next line up.
    iny                ;point at next line down.
    cpy                ;see if at last line in card.
    bmi                ;if not, do more lines.
    pla                ;recover the byte to fill in.
    sta                ;store the byte at the bottom line .
    rts

```

```
.if (0)
```

Roll8bytesOut

Synopsis: Rotates 8 bytes through a, used in the routines to empty the first 640 byte print buffer. the effect is to roll a card up 1 line and leave the byte pushed out on top in a.

Called by: -

Pass: a = byte to fill in at bottom of card.
 r0 = pointer to card to roll up 1 line.

Accessed:

Returned:

Returned (probably not useful but consistant):

Destroyed: a, y

```
.endif
```

```
Roll8bytesOut:
```

```
    ldy    #0           ;init the index to the card.
    lda    (r0),y       ;load the top line from the card.
    pha                    ;save the byte .

10$:
    iny                    ;point at next line down (top byte is lost).
    lda    (r0),y       ;load a line from the card.
    dey                    ;point at next line up.
    sta    (r0),y       ;store the byte at the next line up.
    iny                    ;point at next line down.
    cpy    #7           ;see if at last line in card.
    bmi    10$          ;if not, do more lines.
    pla                    ;recover the byte off the top.
    rts
```

```
.page
```

Commodore Specific Routines

.if(0)

SetGraphics (t_SetGraphics)
UnSetGraphics (t_UnSetGraphics)

Synopsis: SetGraphics (t_SetGraphics) sets graphics mode for the Okimate.
UnSetGraphics (t_UnSetGraphics) exits graphics mode.

Called by:

Pass:

Accessed:

Returned:

Returned (probably not useful but consistant):

Destroyed:

.endif

SetGraphics:

```
    lda    #CGPX    ;send character to set graphics mode.  
    jsr    Ciout  
    rts
```

UnSetGraphics:

```
    lda    #ECGPX   ;send character to unset graphics mode.  
    jsr    Ciout  
    rts
```

.page

.if (0)

SendBuff (t_SendBuff)

Synopsis: SendBuff (t_SendBuff) sends the prntblcard out to the serial port.

Called by:

Pass:

Accessed: prntblcard

Destroyed: a, x

.endif

```
SendBuff:
    ldx    #0                ;initialize the count.
10$:
    txa                    ;save count.
    pha
    lda    prntblcard,x    ;get byte to send.
    ora    #$80            ;add to get out of valid ASCII space.
    jsr    Ciout           ;send this byte
    pla
    tax                    ;recover the count.
    inx                    ;point at next byte.
    cpx    #8              ;are we done with all bytes?
    bne    10$             ;if not, continue with sending.
    rts
```

.page

```
.if(0)
```

Greturn (t_Greturn)

Synopsis: Greturn (t_Greturn) does a graphic line-feed +carriage return

Called by:

Pass: nothing
must be in graphics mode!

Accessed: printmode %0xxxxxxx=no autolf , %1xxxxxxx=autolf.

Destroyed: a

```
.endif
```

Greturn:

```
    lda    #CR           ;carriage return.
    jsr    Ciout         ;send it.
    rts
```

```
.if (0)
```

FormFeed (t_FormFeed)

Synopsis: FormFeed(t_FormFeed) exits graphics mode & does a form feed

Called by:

Pass: nothing

Accessed: fmfdtbl

Returned (probably not useful but consistant):

Destroyed: a

```
.endif
```

```

.if (0)

```

Synopsis: Rotates a hi-res bit-mapped card form the 640 byte print buffer pointed at by r3 into the prntblcard 8 byte buffer.

Pass: r3 = address of the card to be operated on.

Returned (probably not useful but consistant):

```
endif
```

```

Rotate:      sei                ;disable any IRQs.
             ldy      #7        ;initialize the index into the card.
10$:         lda      (r3),y     ;get the byte from the card.
             ldx      #7        ;initialize the index into the printable card.
20$:         ror      a          ;get the least significant bit into c.
             rol      prntblcard,x ;shift it into the printable card table.
             dex                ;next bit.
             bpl      20$        ;if not done, store another bit.
             dey            ;next byte.
             bpl      10$        ;if not done, load another byte.
             cli            ;clear interrupt disable bit.
             rts
PRINTEND:    ; last label in Printer driver, for use in saving
             ; the driver to the disk.

```

WarmStart Configuration

Whenever FirstInit is called, such as when GEOS boots, the following table summarizes the complete state of the machine.

Length	Value	Address	Comment
byte	cld #\$30	CPU_DATA	;clear decimal mode ;Set to ALL RAM
StartRamZero - 8A00	0	StartRamZero	;FIRST, clear GEOS RAM area, Global & ;local, to all 0's
Sets up the Commodore hardware, this includes setting the VIC chip RAM bank, and the MOS 6526 CIA chips, for all of the appropriate conditions for running GEOS.			
byte	#\$2F	CPU_DDR	;init. 6510 data direction reg.
LoadB	#\$36	CPU_DATA	;set memory map to have Kernal & IO in
Initialize key scan values to no keys pressed			
byte	0	cia1base+\$3	;clear cia1 DDRB
byte	0	cia1base+\$F	;clear cia1crb
byte	0	cia2base+\$F	;clear cia2crb
;if NTSC, use \$00, if PAL use \$80			
byte	\$00/\$80	cia1base+\$E	;clear cia1cra & set 50/60 hz bit
byte	\$00/\$88	cia2base+\$E	;clear cia2cra & set 50/60 hz bit
byte	(cia2base) ^ #\$30 #4 GRBANK2 cia2base		;Keep old serial bus data (so we ;don't screw up fast serial bus) set ;graphics chip bank select (CIA port A)
byte	#\$3F	cia2base+\$2	; DDRA direction


```

byte    #$7F    cialbase+$D    ; clear interrupt sources
byte    #$7F    cia2base+$D

vicBase

byte    $00,$00 mob0xpos,mob0ypos ;initial position of object 0
byte    $00,$00 mob1xpos,mob1ypos ;initial position of object 1
byte    $00,$00 mob2xpos,mob2ypos ;initial position of object 2
byte    $00,$00 mob3xpos,mob3ypos ;initial position of object 3
byte    $00,$00 mob4xpos,mob4ypos ;initial position of object 4
byte    $00,$00 mob5xpos,mob5ypos ;initial position of object 5
byte    $00,$00 mob6xpos,mob6ypos ;initial position of object 6
byte    $00,$00 mob7xpos,mob7ypos ;initial position of object 7

byte    $00          msbxpos      ;most significant bits of all
                                   ;cursors x pos

byte    st_den|st_25row|st_bmm|3   ;(Note need y scroll = 3)
        grcntrl1

byte    251          rasreg       ;raster reg (set for interrupt at
                                   ;bottom)

byte    SKIPFLAG     lpxpos       ;(read only flag)
byte    SKIPFLAG     lpypos       ;(read only flag)
byte    %00000001    mobenble    ;(object enable) only the mouse
byte    st_40col      grcntrl2
byte    %00000000    moby2       ;(y - expand)

byte    ((([COLOR_MATRIX]<<2)& $F0)|([SCREEN_BASE>>2]&$0E)
        grmempttr          ;(memory)

byte    %00001111    grirq        ;(interrupt register)
byte    %00000001    grirqen     ;(interrupt enable) enable raster
byte    $00          mobprior    ;(object/background priority)0=obj
byte    $00          mobmcm      ;(object multicolor) 1 = mcm
byte    %00000000    mobx2       ;(x-expand) expand gram cursor

RestoreVectors    ; restore C64 vectors at $0314 from

ROM

byte    0            currentMode  ;Mouse variables

byte    0            no presses to handle (pressFlag)
byte    %11000000    displayBufferOn
word    0            mouseXPosition
byte    0            mouseYPosition
byte    0            mouseOn
word    mousePicData mousePicture

```

```

byte    0                windowTop
byte    199              windowBottom
word    0                leftMargin
word    319              rightMargin
byte    -1               diskData        ;pass release
byte    0                mouseLeft       ;left constraint
byte    0                mouseLeft+1
byte    0                mouseTop        ;top constraint
word    SCREEN_PIXEL_WIDTH-1
                        mouseRight       ;right constraint
byte    SCREEN_PIXEL_HEIGHT-1
                        mouseBottom      ;bottom constraint
byte    #MAXIMUM_VELOCITY
                        maximumMouseSpeed
                        ; get set value for maximumMouseSpeed
byte    #MINIMUM_VELOCITY
                        minimumMouseSpeed
                        ; get set value for min. Mouse speed
byte    #MOUSE_ACCELERATION
                        mouseAcceleration
                        ;get value for acceleration
63 bytes
      error pic        mousePicData

1000 bytes
      #dkgrey<<4|ltgrey
                        COLOR_MATRIX    ;dark grey on light grey screen

byte    #blue            mob0clr        ; mouse color
byte    #blue            mob1clr        ; string cursor color
byte    #black           extclr         ; border color is black

                        ; Then copy arrow picture to mouse

byte    #$08             interleave     ; Set disk interleave to 8

                        ;Initialize disk drive with SetDevice

byte    8                curDrive
byte    8                curDevice      ;reinitialized
byte    1                numDrives      ;Change # of drives to 1

                        Init the time of day clock

byte    cialcrb & #$7F
                        cialcrb         ;get control reg b.
                        ;set to tod clock reads and writes.
                        ;and restore.

```

```

byte    #$83      cialtodh      ;CURRENT HOUR
byte    0          cialtodmin    ;minutes,
byte    0          cialtodsec    ;seconds,
byte    0          cialtod10ths  ;and 1/10 seconds.
byte    0          minutes
byte    0          seconds

```

```

;THE FOLLOWING sets up initial
;Year/Month/Day/Hour, for now, so that
;the disk date stamps look reasonable.

```

```

byte    #86        year          ;CURRENT YEAR
byte    #9          month        ;CURRENT MONTH
byte    #20         day          ;CURRENT DAY
lda     #12         hour         ;CURRENT HOUR (noon)

```

```

;init the various alarm flags,
;vectors.

```

```

byte    0          alarmSetFlag
byte    0          alarmCount
byte    0          alarmTmtVector
byte    0          alarmTmtVector+1

```

```

;Forcefully exit any turbo code
;running

```

```

word    0          applicationMain
word    0          o_InterruptMain
word    0          interruptTopVector
word    0          interruptBottomVector
word    0          mouseVector
word    0          keyVector
word    0          inputVector
word    0          mouseFaultVector
word    0          otherPressVector
word    0          StringFaultVector
word    0          alarmTmtVector
word    0          o_Panic      vector for BRK instruction (BRKVector)
word    0          o_RecoverRectangle

```

```

RecoverVector      ;vector for recover background

```

```

byte    SELECTION_DELAY
byte    0          selectionFlash
byte    0          alphaFlag
byte    ST_FLASH
byte    0          set default to flash (iconSelFlag)
byte    0          faultData

```

```

byte    0          numberOfProcesses

```

```
byte    0          numberAsleep
byte    0          curIconIndex (not selecting icon)
```

Initialize pointers to sprite picture data For more info on how these pointers work see the Commodore 64 Programmer's Reference Manual. Basically, the video space is 16K thus needing only 14 bits to address the entire space. The sprites pictures use 63 bytes and must be on 64 byte boundaries, thus the start of each sprite pic. has an addr. with the low 6 bits 0. Thus $14 - 6 = 8$, only one byte is needed to specify the start address of a picture anywhere in the 16K graphics memory space.

```
byte    [(spr0pic>>6)    spr0pic
byte    [(spr1pic>>6)    spr1pic
byte    [(spr2pic>>6)    spr2pic
byte    [(spr3pic>>6)    spr3pic
byte    [(spr4pic>>6)    spr4pic
byte    [(spr5pic>>6)    spr5pic
byte    [(spr6pic>>6)    spr6pic
byte    [(spr7pic>>6)    spr7pic
```

```
Grey the Screen:          ;place a grey pattern all over the screen
                          ;A000 to BF3F
```

```
MoveShortBlock $FD30,$0314,32
                          ;Restore the C64 vectors in page 3
                          ;from ROM
```

mike 31:

Appendix A.

Constants

TRUE	=	-1
FALSE	=	0

The following equates define the numbers written to the CPU_DATA register (location 1 in c64). These numbers control the hardware memory map of the c64.

IO_IN	=	\$35	;60K RAM, 4K I/O space in
RAM_64K	=	\$30	;64K RAM
KRNL_BAS_IO_IN	=	\$37	;both Kernal and basic ROM's mapped into memory
KRNL_IO_IN	=	\$36	;Kernal ROM and I/O space mapped in

Constants for misc

PROMPT_DELAY	=	60	;note maximum value is 63
VERTICAL_SPACE	=	2	
HORIZONTAL_SPACE	=	4	
;			
POSITIVE	=	%00000000	
NEGATIVE	=	%10000000	

Menu

```

MAXIMUM_MENU_ITEMS      =      15
MAXIMUM_MENU_NESTING    =      4
DATA_BUFFER_SIZE        =      8
PUTCHAR_BUFFER_SIZE     =      8

HORIZONTAL      =      %00000000
VERTICAL        =      %10000000
CONSTRAINED     =      %01000000
UN_CONSTRAINED  =      %00000000

SELECTION_DELAY =      10      ;Delay between inversions of menu selection
                                ;1/6 second

                                ;Offsets to variables in the menu structure

OFF_M_Y_TOP      =      0      ;offset to y position of top of menu
OFF_M_Y_BOT      =      1      ;offset to y position of bottom of menu
OFF_M_X_LEFT     =      2      ;offset to x position of left side of menu
OFF_M_X_RIGHT    =      4      ;offset to x position of right side of menu
OFF_NUM_M_ITEMS  =      6      ;offset to Alignment|Movement|Number if items
OFF_1ST_M_ITEM   =      7      ;offset to record for 1st menu item in
                                ;structure

SUB_MENU         =      $80     ;for setting byte in menu table that indicates
DYNAMIC_SUB_MENU =      $40     ;whether menu item causes action or submenu
MENU_ACTION      =      $00

```

Processes

```

MAXIMUM_PROCESSES      =      20
SLEEP_MAXIMUM          =      20

                                ; _____
                                ;Possible values for processFlags

;
SET_RUNABLE           =      %10000000      ;runable flag
SET_BLOCKED           =      %01000000      ;process blocked flag
SET_FROZEN            =      %00100000      ;process frozen flag
SET_NOTIMER           =      %00010000      ;not a timed process flag
;
RUNABLE_BIT           =      7              ;runable flag

```

```

BLOCKED_BIT      =      6      ;process blocked flag
FROZEN_BIT       =      5      ;process frozen flag
NOTIMER_BIT      =      4      ;not a timed process flag
;

```

Text

```

;_____
;Bit flags in mode

```

```

SET_UNDERLINE    =      %10000000
SET_BOLD         =      %01000000
SET_REVERSE      =      %00100000
SET_ITALIC       =      %00010000
SET_OUTLINE      =      %00001000
SET_SUPERSCRIPT  =      %00000100
SET_SUBSCRIPT    =      %00000010
SET_PLAINTEXT    =      0

```

```

UNDERLINE_BIT    =      7
BOLD_BIT         =      6
REVERSE_BIT      =      5
ITALIC_BIT       =      4
OUTLINE_BIT      =      3
SUPERSCRIPT_BIT  =      2
SUBSCRIPT_BIT    =      1

```

```

;_____
;PutChar Constants

```

```

;
EOF              =      0      ;end of text object
NULL             =      0      ;end of string
BACKSPACE        =      8      ;move left a card
TAB              =      9
FORWARDSPACE     =      9      ;move right one card
LF               =      10     ;move down a card row
HOME             =      11     ;move to left top corner of screen
UPLINE           =      12     ;move up a card line
PAGE_BREAK       =      12     ;page break
CR               =      13     ;move to beginning of next card row
UNDERLINEON      =      14     ;turn on underlining
UNDERLINEOFF     =      15     ;turn off underlining
ESC_GRAPHICS     =      16     ;escape code for graphics string
ESC_RULER        =      17     ;ruler escape
REVERSEON        =      18     ;turn on reverse video
REVERSEOFF       =      19     ;turn off reverse video
GOTOX            =      20     ;use next byte as 1+x cursor
GOTOY            =      21     ;use next byte as 1+y cursor
GOTOXY           =      22     ;use next bytes as 1+x and 1+y cursor

```

NEWCARDSET	=	23	;use next two bytes as new font id
BOLDON	=	24	;turn on BOLD characters
ITALICON	=	25	;turn on ITALIC characters
OUTLINEON	=	26	;turn on OUTLINE characters
PLAINTEXT	=	27	;plain text mode
USELAST	=	127	;erase character
SHORTCUT	=	128	;shortcut character
<hr/>			
RULER_SIZE	=	27	;size of a ruler escape

Keyboard

KEY_QUEUE_SIZE	=	16	;Size of keyboard queue
KEY_REPEAT_COUNT	=	15	;1/4 second. Repeat delay for keyboard
			;auto-repeat (max is 254, NOT 255)
			;These are values for keys
KEY_INVALID	=	31	
KEY_F1	=	1	
KEY_F2	=	2	
KEY_F3	=	3	
KEY_F4	=	4	
KEY_F5	=	5	
KEY_F6	=	6	
KEY_F7	=	14	
KEY_F8	=	15	
KEY_UP	=	16	
KEY_DOWN	=	17	
KEY_HOME	=	18	
KEY_CLEAR	=	19	
KEY_LEFTARROW	=	20	
KEY_UPARROW	=	21	
KEY_STOP	=	22	
KEY_RUN	=	23	
KEY_BPS	=	24	
KEY_LEFT	=	BACKSPACE	
KEY_RIGHT	=	30	
KEY_DELETE	=	29	
KEY_INSERT	=	28	

Mouse

MOUSE_ACCELERATION	=	127	;acceleration of mouse
MAXIMUM_VELOCITY	=	127	;maximum velocity
MINIMUM_VELOCITY	=	30	;minimum velocity

;Bit flags in mouseOn

```

SET_MOUSEON      =      %10000000
SET_MENUON       =      %01000000
SET_ICONSON      =      %00100000
;
MOUSEON_BIT      =      7
MENUON_BIT       =      6
ICONSON_BIT      =      5

```

Graphics

;----Constants for screen size

```

SCREEN_BYTE_WIDTH      =      40
SCREEN_PIXEL_WIDTH     =      320
SCREEN_PIXEL_HEIGHT    =      200
SCREEN_SIZE            =      SCREEN_BYTE_WIDTH * SCREEN_PIXEL_HEIGHT

```

;
;Bits used to set displayBufferOn flag
;(controls which screens get written.

```

ST_WR_FORE           =      $80
ST_WR_BACK           =      $40
ST_WRGS_FORE         =      $20

```

;write to foreground
;write to background

;g-string only writes to foreground.
;Values for graphics strings

```

MOVEPENTO           =      1
LINETO              =      2
RECTANGLETO         =      3
;PENFILL            =      4
NEWPATTERN          =      5
ESC_PUTSTRING       =      6
FRAME_RECTO         =      7
PEN_X_DELTA         =      8
PEN_Y_DELTA         =      9
PEN_XY_DELTA        =      10

```

;move pen to x,y
;draw line to x,y
;draw a rectangle to x,y
;fill with the current pattern
;set a new pattern
;start putstring interpretation
;do a frame rectangle thing
;move pen by signed word delta in x
;move pen by signed word delta in y
;move pen signed word delta in x & y

;
;VIC colors

```

black               = 0
white               = 1
red                 = 2
cyan                = 3
purple              = 4
green               = 5
blue                = 6
yellow              = 7
orange              = 8
brown               = 9
ltred               = $A
dkgrey              = $B
grey                = $C

```

```
medgrey      = grey
ltgreen      = $D
ltblue       = $E
ltgrey       = $F
```

```
;_____
;Values for PutDecimal calls
```

```
SET_LEFTJUSTIFIED    =    %10000000    ;left justified
SET_RIGHTJUSTIFIED    =    %00000000    ;left justified
SET_SUPPRESS          =    %01000000    ;no leading 0's
SET_NOSUPPRESS        =    %00000000    ;leading 0's
```

Menu

```
CLICK_COUNT    = 30
;number of interrupts to count.
;The following equate is loaded into the RAM variable
;dblClickCount when an icon is first "clicked on".dblClickCount
;is decremented each interrupt, if it is non-zero when the icon
;is again selected, then the double click flag (r0H) is passed
;to the service routine with a value of TRUE. If the
;dblClickCount variable is zero when the icon is clicked on,
;then the flag is passed with a value of FALSE
```

```
;_____
;These equates are bit values for iconSelfFlag
;that determines how an icon selection is
;indicated to the user. If ST_FLASH is set,
;ST_INVERT is ineffective.
ST_FLASH        = $80    ;bit to indicate icon should flash.
ST_INVERT       = $40    ;bit to indicate icon should be inverted.
```

```
;offsets into the icon structure
```

```
OFF_NUM_ICNS    = 0      ;offset from start of icon header to byte
                        ;that specifiys the number of icons in struc.
OFF_ICN_XMOUSE  = 1      ;offset from start of icon header to byte
                        ;that specifiys the mouse x start position
OFF_ICN_YMOUSE  = 3      ;offset from start of icon header to byte
                        ;that specifiys the mouse y start position
```

```
;_____
;offset into an icon record in icon structure.
```

```
OFF_PIC_ICON    = 0      ;offset to picture pointer for icon
OFF_X_ICON_POS  = 2      ;offset to x position of icon
OFF_Y_ICON_POS  = 3      ;offset to y position of icon
OFF_WIDTH_ICON  = 4      ;offset to width of icon
```

OFF_HEIGHT_ICON = 5	;offset to height of icon
OFF_SRV_RT_ICON = 6	;offset to pointer to service routine for icon
OFF_NXT_ICON = 8	;offset to the next icon in icon structure

Flags

```

;
;Values for pressFlag
;
KEYPRESS_BIT    =      7      ;other keypress
INPUT_BIT       =      6      ;input device change
MOUSE_BIT       =      5      ;mouse press

SET_KEYPRESS    =      %10000000 ;other keypress
SET_INPUTCHANGE =      %01000000 ;input device change
SET_MOUSE       =      %00100000 ;mouse press

;
;Values for faultFlag
;
OFFTOP_BIT      =      7      ;mouse fault up
OFFBOTTOM_BIT   =      6      ;mouse fault down
OFFLEFT_BIT     =      5      ;mouse fault left
OFFRIGHT_BIT    =      4      ;mouse fault right
OFFMENU_BIT     =      3      ;menu fault

;
SET_OFFTOP      =      %10000000 ;mouse fault up
SET_OFFBOTTOM   =      %01000000 ;mouse fault down
SET_OFFLEFT     =      %00100000 ;mouse fault left
SET_OFFRIGHT    =      %00010000 ;mouse fault right
SET_OFFMENU     =      %00001000 ;menu fault

;
;
ANY_FAULT = SET_OFFTOP|SET_OFFBOTTOM|SET_OFFLEFT|SET_OFFRIGHT|SET_OFFMENU
;
;

```

GEOS FILE TYPES

```

;
;This is the value in the "GEOS file type"
;byte of a dir entry that is pre-GEOS:
NOT_GEOS      = 0      ;Old c64 file, without GEOS header
;                (PRG, SEQ, USR, REL)

```

;The following are GEOS file types reserved for compatibility
 ;with old c64 bfiles, that have simply had GEOS header placed

;on them. Users should; be able to double click on files of
;type BASIC and ASSEMBLY, whereupon ;they will be fast-loaded
;and executed (from under BASIC)

BASIC	= 1	;c64 BASIC program, with a GEOS header attached. ; (Commodore file type PRG) To be used on ; programs that were executed before GEOS ; with: ; LOAD "FILE",8 ; RUN
ASSEMBLY	= 2	;c64 ASSEMBLY program, with a GEOS header attached. ; (Commodore file type PRG) To be used on ; programs that were executed before GEOS ; with: ; LOAD "FILE",8,1 ; SYS(Start Address)
DATA	= 3	;Nonexecutable DATA file (PRG, SEQ, or USR) with a ; GEOS header attached for icon & notes ability

;The following are file types for GEOS applications & system use:
;ALL files having one of these GEOS file types should be of
;Commodore file type USR.

SYSTEM	= 4	;GEOS system file
DESK_ACC	= 5	;GEOS desk accessory file
APPLICATION	= 6	;GEOS application file
APPL_DATA	= 7	;data file for a GEOS application
FONT	= 8	;GEOS font file
PRINTER	= 9	;GEOS printer driver
INPUT_DEVICE	=10	;INPUT device (mouse, etc.)
DISK_DEVICE	=11	;DISK device driver
SYSTEM_BOOT	=12	;GEOS system boot file (for GEOS, GEOS BOOT, ; GEOS KERNAL)
TEMPORARY	=13	;Temporary file type, for swap files. The deskTop ; will automatically delete all files of this ; type upon opening a disk. ;when creating additionally swap files , ;use the GEOS TEMPORARY type and start the filename ;with the first character being PLAINTEXT. Example: ;swapName: ; .byte PLAINTEXT,"My swap file",0 this way it ;This will prevent accidentally overwriting a user ;file with the same name ("My swap file"). -mfarr
AUTO_EXEC	=14	;Application to automatically be loaded & run just ; after booting, but before deskTop runs.
NUM_FILE_TYPES	=15	; # of file types, including NON_GEOS (=0) ;GEOS file structure types. ;Each "structure type" specifies the organization of ;data blocks on the disk, and has nothing to do with ;the data in the blocks.

```

SEQUENTIAL      = 0      ;standard T,S structure (like commodore SEQ
                        ;      and PRG files)
VLIR             = 1      ;variable-length-indexed-record file (used for
                        ;Fonts, Documents & some programs)
                        ;This is a GEOS only format.

```

Standard Commodore File Types

;i.e., the file types supported by the old 1541 DOS.

```

DEL      = 0      ;DELETED file
SEQ      = 1      ;SEQuential file
PRG      = 2      ;PROGram file
PROG     = 2      ;PROGram file
USR      = 3      ;USER file
USER     = 3      ;USER file
REL      = 4      ;RELAtive file

TOTAL_BLOCKS = 664 ;number of blocks on disk, not including dir. track

```

Directory Header

```

OFF_TO_BAM      = 4      ;offset to first BAM entry

OFF_DISK_NAME   = 144    ;offset in directory header to disk name string

OFF_OP_TR_SC    = 171    ;offset to track and sector for off page directory
                        ;entries (1 block,i.e.,8 files may be moved off page)

OFF_GEOS_ID     = 173    ;position in directory header where GEOS ID string
                        ;is located

OFF_GEOS_DTYPE  = 189    ;Position where GEOS disk type is. Currently, is
                        ;0 for normal disk, 'B' for BOOT disk.
                        ;Location is zeroed on destination disk copies

```

Directory Entry

;Offsets within a Directory Entry. (a specific file's directory entry)

```

;This bit equate is used to test and/or set the write
;protect bit in the commodore file type byte in file's
;directory entry.

```

```

ST_WR_PR      = $40      ;write protect bit, bit 6 of byte 0 in the dir. entry.

OFF_CFILE_TYPE = 0      ;offset to standard Commodore file type indicator
OFF_INDEX_PTR  = 1      ;offset to Index table pointer (VLIR file)
OFF_DE_TR_SC   = 1      ;offset in entry to track for file's 1st data block
FRST_FILE_ENTRY = 2      ;offset in a directory block to the first file entry
OFF_FILE_NAME  = 3      ;offset to file name in file's directory entry
OFF_GHDR_PTR   = 19      ;offset to track and sector info on where header
                        ;block is located
OFF_GSTRUCT_TYPE = 21    ;offset to GEOS file structure type
OFF_GFILE_TYPE  = 22      ;offset to geos file type indicator
OFF_FILE_YEAR   = 23      ;offset to year (1st byte of date stamp)
OFF_FILE_SIZE   = 28      ;offset to size of the file in blocks.
OFF_NXT_FILE    = 32      ;offset to next file entry in directory structure

```

.page

File Header

;offsets into a GEOS file header block

```

OFF_GHICN_WIDTH = 2      ;byte, indicates width in bytes of file icon
OFF_GHICN_HEIGHT = 3     ;byte, indicates height in lines of file icon
OFF_GHICN_PIC   = 4      ;64 bytes, picture data for file icon
OFF_GHCMDR_TYPE = 68      ;byte, Commodore file type for directory entry
OFF_GHGEOS_TYPE = 69      ;byte, GEOS file type for directory entry
OFF_GHSTRUCT_TYPE = 70    ;byte, GEOS File structure type
OFF_GHSTRT_ADDR = 71      ;2 bytes, start address in memory where file saved from
OFF_GHEND_ADDR  = 73      ;2 bytes, end address of file in memory
OFF_GHSTRT_VEC  = 75      ;2 bytes, initialization vector if file is an appl.
OFF_GHFNNAME    = 77      ;20 bytes, permanent filename.
OFF_GHFONTID    = 128     ;In the header file for a font, offset to the 10 bit
                        ;ID number of the font.
OFF_GHPOINT_SIZES = 130   ;Offset into File Header for font file, points to list
                        ;of point sizes for this font. Max 16 point sizes
OFF_GHSET_LENGTHS = 97    ;For font files. Size (in bytes) of each character set
                        ;listed in the GHPOINT_SIZE tbl. 2 bytes * num_pt_sizes
OFF_GHP_DISK     = 97      ;20 bytes, disk name of parent application's disk
                        ;(only if file is application data) last 4 chars are
                        ;null, chars 13-15 are V1.3
OFF_GHP_FNAME    = 117     ;20 bytes, permanent filename of parent application
                        ;(only if file is application data)
OFF_GH_AUTHOR    = 97      ;offset to author's name in header file.
                        ;same as parent disk name, 2 never used in same file.
OFF_GHINFO_TXT   = $A0     ;offset to notes that are stored with the file and
                        ;edited in the get info box.

```

GetFile Equates

;The following equates define file loading options for the GEOS DOS GetFile
;routine and assorted Ld routines. These bit definitions are used to
;set the RAM variable loadOpt

ST_LD_AT_ADDR	= 01	;Load At Address, load file at caller specified ;address instead of address file was saved from
ST_LD_DATA	= \$80	;(application files only) data file was opened ;data file usually passed in r3 at same time ;Application should load data file
ST_PR_DATA	= \$40	;(application files only) application was invoked ;to print a specified data file. r3 has data filename

.page

Disk

;Low-level disk routines

N_TRACKS	= 35	;# of tracks available on the 1541 disk
DIR_TRACK	= 18	;track # reserved on disk for directory
; EQUATES for variable "driveType"		
; High two bits of driveType have special ; meaning (only 1 may be set):		
; Bit 7: if 1, then RAM DISK		
; Bit 6: if 1, then Shadowed disk		
DRV_NULL	= 0	; No drive present at this device address
DRV_1541	= 1	; Drive type Commodore 1541
DRV_1571	= 2	; Drive type Commodore 1571
DK_NM_ID_LEN	=18	; # of characters in disk name
;Disk access commands		
MAX_CMND_STR	= 32	;maximum length a command string would have.
DIR_ACC_CHAN	= 13	;default direct access channel
REL_FILE_NUM	= 9	;logical file number & channel used for ;relative files.
CMND_FILE_NUM	= 15	;logical file number & channel used for ;command files
;indexes to a command buffer for setting ;the track and sector number for direct access ;command		
TRACK	= 9	;offset to low byte decimal ASCII track number
SECTOR	= 12	;offset to low byte dec. ASCII sector number

[illegible]


```

STRUCT_MISMATCH = 10
    ; This error occurs when a specific-structure file routine
    ; is requested to perform an operation on a file that has
    ; a different structure type.

BUFFER_OVERFLOW = 11
    ; This error occurs during a call to ReadFile when the # of
    ; bytes in the file being loaded is larger than the maximum #
    ; allowed, as specified by the calling application

.page

CANCEL_ERR      = 12
    ;this error is here to allow the programmer to have a
    ;dialog box at some lower level of a disk operation, and
    ;if the user hits the CANCEL icon, to escape out of the
    ;operation using the normal disk error escape route.
    ;i.e., after detecting the cancel selection, load x with
    ;CANCEL_ERR and return. The convention for all disk error
    ;handling is to pass the error in x and return. The top level
    ;routine that initiated the operation then checks for the
    ;error.

DEVICE_NOT_FOUND= 13
    ;this error occurs whenever the c64 Kernal routine "Listen"
    ;is used to attempt to open channels to a device on the
    ;serial bus, and a Device Not Found error is returned.

HDR_BLK_NOT_THERE = $20
    ;this error occurs if the header block cannot be found

NO_SYNC         = $21
    ;error: occurs if drive can't find sync mark on disk,
    ;typically causes: disk not in drive, drive door not shut,
    ;unformatted disk.

DATA_BLK_NOT_THERE = $22
    ;error: data block not present

DATA_CHKSUM_ERR = $23
    ;error: data block checksum error (bad block on disk)

WR_VER_ERR      = $25
    ;error: verify operation after write failed. typical cause:
    ;bad disk block, screwed up drive (bad luck)

WR_PR_ON        = $26
    ;error: attempt to write to a disk that has a write protect
    ;tab over its notch.

HDR_CHKSUM_ERR  = $27
    ;error: checksum error in header block

DSK_ID_MISMATCH = $29
    ;error: ID read from disk does not match internal ID expected
    ;for this disk. typical cause: disk changed without issuing a
    ;new disk command, usually user error, illegally switching
    ;disk.

BYTE_DECODE_ERR = $2E
    ;can't decode flux transitions off of disk

DOS_MISMATCH    = $73
    ;error: wrong DOS indicator on the disk.

```

DIALOG BOXES

```
DEF_DB_POS      = $80    ;bit set to indicate default dialogue box position
SET_DB_POS      = 0      ;if MSB is clear, user must provide the position
```

```
;Dialog box descriptor table commands
```

```
;System icons available for use in dialog boxes. These icons are predefined
;and can make dialog box construction simple. Most of these icons cause
;the dialog box to be exited and control returned to the application with
;the dialog command that was selected passed in sysDBData
```

```
;NOTE: in all dialog box commands, the positions specified are offsets
;from the upper left corner of the dialog box. Usually these offsets are
;in pixels, but occasionally the x offset is in bytes (as in case of icons).
```

```
OK              =      1      ;put up system icon for "OK", command is followed by
                               ;2 byte position indicator, x pos. in bytes, y pos.
                               ;in pixels. NOTE: positions are offsets from the top
                               ;left corner of the dialog box.
CANCEL          =      2      ;like OK, system dialog box icon, position follows
                               ;command
YES             =      3      ;like OK, system dialog box icon, position follows
                               ;command
NO              =      4      ;like OK, system dialog box icon, position follows
                               ;command
OPEN            =      5      ;like OK, system dialog box icon, position follows
                               ;command
DISK            =      6      ;like OK, system dialog box icon, position follows
                               ;command
FUTURE1         =      7      ;reserved for future system icons
FUTURE2         =      8      ;reserved for future system icons
FUTURE3         =      9      ;reserved for future system icons
FUTURE4         =     10      ;reserved for future system icons
```

```
;More dialog box descriptor table commands
```

```
DBTXTSTR =      11      ;command to display a text string. Command is followed
                        ;by address of string and start position of string.
DBVARSTR =      12      ;used to put out variant strings, command is followed
                        ;by the address of the zpage register that points
                        ;to the string to be output. This byte is followed by
                        ;the position bytes.
```

```

DBGETSTRING = 13      ;Get an ASCII string from teh user into a buffer, the
                      ;position to place the string follows the command,
                      ;followed by the zpage offset to the reg. that points
                      ;to the input string buffer, followed by the maximum
                      ;number of chars to read in.

DBSYSOPV = 14         ;command has no arguments, any press not over an icon
                      ;causes a return to the application.

DBGPHSTR = 15         ;command to cause graphics string to be executed. The
                      ;command is followed by a pointer to the graphics str.

DBGETFILES = 16       ;This command will cause the dialog box
                      ;structure to position a box on the left side of the dialog
                      ;box that contains a list of filenames for all files
                      ;that match the passed type. If more types are on the
                      ;current disk than can be listed in the box, the
                      ;other files may be seen by pressing scroll arrows
                      ;that appear under the box. the command is followed
                      ;by the top left corner to place the box in. It is expected
                      ;on entry to DoDlgBox that r7L has the file type, r5 points
                      ;at buffer to return selected filename, and r10 is null if
                      ;all files matching the type are to be listed and a pointer
                      ;to a permanent name string if only files matching the perm.
                      ;name are to be listed.

DBOPVEC = 17          ;user defined other press vector, routine address
                      ;that follows the call is set as the otherPressVector.

DBUSRICON = 18         ;user defined icon, icon pointed at by following word
                      ;is added to the dialogue box icon structure. The
                      ;position for the icon follows the pointer

DB_USR_ROUT = 19      ;This command is the "Do It Yourself" command. The
                      ;address of the routine to call follows the command. This
                      ;routine will be called like a subroutine when the dialog
                      ;box is being put up.

```

;The following equates are used to specify offsets into a dialog box
;descriptor table.

```

OFF_DB_FORM      = 0      ;offset to box form description, i.e., shadow or not
OFF_DB_TOP       = 1      ;offset to position for top of dialog box
OFF_DB_BOT       = 2      ;offset to position for bottom of dialog box
OFF_DB_LEFT      = 3      ;offset to position for left of dialog box
OFF_DB_RIGHT     = 5      ;offset to position for right of dialog box
OFF_DB_1STCMND   = 7      ;offset to 1st command in dialog box descriptor table
                      ;if user specifies the position

```

;The following equate defines the maximum number of icons that may be placed
;in the dialog box icon structure. This equate is used to allocate the
;RAM for the dialog box icons.

```

MAX_DB_ICONS     = 8      ;maximum number of icons in a dialog box.

```

;This equate is used to specify the x and y offset for the dialogue box
 ;position to draw the shadow box for the dialog box if one is requested.
 ;This equate should be a multiple of 8 so that compatibility with future
 ;color versions may be maintained.

DB_SHAD_OFFSET = 8 ;position offset for shadow box to dialog box.

;The following equates specify the dimensions of the system defined dialog
 ;box icons.

SYS_DB_ICN_WIDTH = 6 ;width in bytes
 SYS_DB_ICN_HEIGHT = 16 ;height in pixels

;THESE EQUATES ARE USED TO SUPPORT THE GET FILE DIALOGUE BOX COMMAND

DB_F_BOX_WIDTH = 124 ;width of filename box that appears inside a dialog
 ;box and holds filenames that match a passed file type
 DB_F_BOX_HEIGHT = 88 ;height of filename box that appears inside a dialog
 ;box and holds filenames that match a passed file type
 DB_F_SCR_ICN_X_OFF = 7 ;offset from left edge of file name box to where
 ;the scroll icon goes (byte offset)
 DB_F_SCR_ICN_HEIGHT = 14 ;height of the scroll icon (off bottom of box)
 DBF_LINE_HEIGHT = 16 ;height (from bottom of the box) to where line that
 ;separates scroll icon area from the filenames goes.
 MAX_DB_FILES = 15 ;maximum number of files that will fit in the
 ;filename buffer (fileTrScTab).
 DB_FSCR_ICN_WIDTH = 3 ;width of icon (in bytes) used to scroll up and down
 ;over the filenames in the Get files box
 DB_FSCR_ICN_HEIGHT = 12 ;height of icon (in pixels) used to scroll up and down
 ;over the filenames in the Get files box
 DBF_Y_LINE_OFF = 14 ;y offset between filename lines
 DB_TXT_BASE_OFF = 9 ;offset from the top of a line's rectangle to the x
 ;position for baseline for text output in the rect.

;MORE DIALOG BOX RELATED EQUATES

;These equates define a standard, default, dialog box position and
 ;size as well as some standard positions within the box for outputting
 ;text and icons.

DEF_DB_TOP = 32 ;top y coordinate of default box
 DEF_DB_BOT = 127 ;bottom y coordinate of default box
 DEF_DB_LEFT = 64 ;left edge of default box
 DEF_DB_RIGHT = 255 ;right edge of default box
 TXT_LN_X = 16 ;standard text x start

```

TXT_LN_1_Y      = 16      ;standard text line y offsets
TXT_LN_2_Y      = 32
TXT_LN_3_Y      = 48
TXT_LN_4_Y      = 64
TXT_LN_5_Y      = 80

DB_ICN_X_0      = 1       ;byte offset to left side standard icon x position
DB_ICN_X_1      = 9       ;byte offset to center standard icon x position
DB_ICN_X_2      = 17      ;byte offset to right side standard icon x position

DB_ICN_Y_0      = 8       ;byte offset to left side standard icon y position
DB_ICN_Y_1      = 40      ;byte offset to center standard icon y position
DB_ICN_Y_2      = 72      ;byte offset to right side standard icon y position

```

VIC Chip

```

GRBANK0 = %11      ;bits indicate VIC RAM is $0000 - $3FFF, 1st 16K
GRBANK1 = %10      ;bits indicate VIC RAM is $4000 - $7FFF, 2nd 16K
GRBANK2 = %01      ;bits indicate VIC RAM is $8000 - $BFFF, 3rd 16K
GRBANK3 = %00      ;bits indicate VIC RAM is $c000 - $FFFF, 4th 16K

SKIPFLAG = $aa     ;flag used to indicate table entry should be
                   ;skipped (e.g., in initVic in powerUp)

MOUSE_SPR_NUM = 0   ;sprite number used for mouse (used to set VIC)

VIC_Y_POS_OFFSET = 50 ;position offset from 0 to position a hardware
                     ;sprite at the top of the screen. Used to map from
                     ;GEOS coordinates to hardware position coordinates.

VIC_X_POS_OFFSET = 24 ;as above, offset from hardware 0 pos. to left of
                     ;screen, used to map GEOS coordinates to VIC.

ALARM_MASK      =      %00000100      ;mask for the alarm bit in the
                                   ;CIA chip int cntrl reg.

.page

```

Desk Accessory Save Foreground Bit

```

FG_SAVE      =      %10000000      ;save and restore foreground graphics data.
CLR_SAVE      =      %01000000      ;save and restore color information.
;

```



Appendix B.

Global GEOS Variables

This file contains a description of the GEOS memory map and an allocation of zpage memory and system RAM used by the operating system. Several constants are referred to in this file, they are defined in the Appendix C.

GEOS MEMORY MAP

Num. Bytes Decimal	Address Range Hexadecimal	Description
1	0000	6510 Data Direction Register
2	0001	6510 I/O register
110	0002-006F	zpage used by GEOS and application
16	0070-007F	zpage for only application, regs a2-a9
123	0080-00FA	zpage used by c64 Kernal & BASIC
4	00FC-00FE	zpage for only applications, regs a0-a1
1	00FF	Used by Kernal ROM & BASIC routines
256	0100-01FF	6510 stack
512	0200-03FF	RAM used by c64 Kernal ROM routines
23552	0400-5FFF	Application program and data
8000	6000-7F3F	Background screen RAM
192	7F40-7FFF	Application RAM
2560	8000-89FF	GEOS disk buffers and variable RAM
512	8A00-8BFF	Sprite picture data
1000	8C00-8FD7	Video color matrix
16	8FD8-8FF7	GEOS RAM
8	8FF8-8FFF	Sprite pointers
4096	9000-9FFF	GEOS code
8000	A000-BF3F	Foreground screen RAM or BASIC ROM
192	BF40-BFFF	GEOS tables
4288	C000-CFFF	4k GEOS Kernal code, always resident
4096	D000-DFFF	4k GEOS Kernal or 4k c64 I/O space
7808	E000-FE74	8k GEOS Kernal or 8k c64 Kernal ROM
378	FE80-FFF9	Input driver
6	FFFA-FFFF	6510 NMI, IRQ, and reset vectors

;C64 memory locations

```

CPU_DDR      =      $0000    ;address of 6510 data direction register
CPU_DATA     =      $0001    ;address of 6510 data register
curDevice    =      $00BA    ;current serial device #
irqvec       =      $0314    ;irq vector (two bytes).
bkvec        =      $0316    ;break ins vector (two bytes).
nmivec       =      $0318    ;nmi vector (two bytes).
kernalVectors =      $031A    ;location of Kernal vectors
vichbase     =      $D000    ;video interface chip base address.
sidbase      =      $D400    ;sound interface device base address.
ctab         =      $D800

```



```

cialbase      =      $DC00    ;1st communications interface adaptor (CIA).
cia2base      =      $DD00    ;second CIA chip
kernalList    =      $FD36

```

;Addresses of c64 Kernal Routines, for information on their use see
;pages 272 - 306 in the Commodore 64 Programmer's Reference Guide.

```

Acptr  =      $FFA5    ;accept a character from the serial bus
Chkin  =      $FFC6    ;open channel associated with logical file for input
Chkout =      $FFC9    ;open channel associated with logical file for output
Chrin  =      $FFCF    ;get byte from serial channel
Chrout =      $FFD2    ;output a byte to the serial channel
Ciout  =      $FFA8    ;send a byte over the serial bus
Cint   =      $FF81    ;init. screen editor, 6567 (used to go to Basic)
Clall  =      $FFE7    ;close all open files
Close  =      $FFC3    ;close logical file passed in acc.
Clrchn =      $FFCC    ;clear I/O channels
Getin  =      $FFE4    ;get chars from c64 keyscan (not used)
Iobase =      $FFF3    ;returns address of I/O page in mem. (not used)
Ioinit =      $FF84    ;init. all I/O devices
Listen =      $FFB1    ;command a device on the serial bus to listen
Load   =      $FFD5    ;load data from input device into C64 memory.
Membot =      $FF9C    ;set or read bottom of memory
Memtop =      $FF99    ;set or read top of memory
Open   =      $FFC0    ;open a logical file
;
;      DON'T EVER USE THIS INCREDIBLY STUPID ROUTINE!
;      (Writes to $FD30 as well as $0314)
;Restor =      $FF8A    ;restore default c64 exec interrupt & system vectors
Save    =      $FFD8    ;save memory to device
Scnkey  =      $FF9F    ;keyboard scanning routine.
Second  =      $FF93    ;send secondary address for listen.
Setlfs  =      $FFBA    ;set up a logical file
Setmsg  =      $FF90    ;manages Kernal message handling
Setnam  =      $FFBD    ;set up filename
Tksa    =      $FF96    ;send a secondary address to device commanded to talk
Unlsn   =      $FFAE    ;send unlisten command to all devices on serial bus
Untlk   =      $FFAB    ;send untalk command to all devices on serial bus.

```

;GEOS memory space definitions

```

SYSTEM_RAM      =      $0400
APPLICATION_RAM  =      $0400    ; start of application space
BACK_SCREEN_BASE =      $6000    ; base of background screen
PRINTBASE       =      $7900    ; load address for print
; drivers
APPLICATION_VAR  =      $7F40    ; application variable space
OS_VARS         =      $8000    ; OS variable base
SPRITE_PICS     =      $8A00    ; base of sprite pictures
COLOR_MATRIX    =      $8C00    ; video color matrix
SCREEN_BASE     =      $A000    ; base of foreground screen
OS_ROM          =      $C000    ; start of OS code space
OS_JUMPTAB      =      $C100    ; start of GEOS jump table
MOUSE_BASE      =      $FE80    ; start of input driver

```

;Equates to permanently fixed variables in the \$C000 page

bootName	=	\$C006	; start of "GEOS BOOT: string
version	=	\$C00F	; GEOS version byte
nationality	=	\$C010	; nationality byte
dateCopy	=	\$C018	; copy of year, month, day

;Jump addresses within print drivers

InitForPrint	=	PRINTBASE	; \$7900 address of InitForPrint
StartPrint	=	PRINTBASE+3	; \$7903 address of StartPrint entry
PrintBuffer	=	PRINTBASE+6	; \$7906 address of PrintBuffer entry
StopPrint	=	PRINTBASE+9	; \$7909 address of StopPrint entry
GetDimensions	=	PRINTBASE+12	; \$7912 address of GetDimensions
PrintASCII	=	PRINTBASE+15	; \$7915 address of PrintASCII entry
StartAscii	=	PRINTBASE+18	; \$7918 address of StartASCII entry
SetNLQ	=	PRINTBASE+15	; \$7915 address of SetNLQ entry

;Jump addresses within input drivers

InitMouse	=	MOUSE_BASE	; \$FE80 address of InitMouse entry
SlowMouse	=	MOUSE_BASE+3	; \$FE83 address of SlowMouse entry
UpdateMouse	=	MOUSE_BASE+6	; \$FE86 address of UpdateMouse entry

;Addresses of specific sprite picture data

spr0pic	=	SPRITE_PICS	; \$8A00 addr. of sprite pic. data
spr1pic	=	spr0pic+64	; \$8A40 addr. of sprite pic. data
spr2pic	=	spr1pic+64	; \$8A80 addr. of sprite pic. data
spr3pic	=	spr2pic+64	; \$8C00 addr. of sprite pic. data
spr4pic	=	spr3pic+64	; \$8B00 addr. of sprite pic. data
spr5pic	=	spr4pic+64	; \$8B40 addr. of sprite pic. data
spr6pic	=	spr5pic+64	; \$8B80 addr. of sprite pic. data
spr7pic	=	spr6pic+64	; \$8BC0 addr. of sprite pic. data
NMI_VECTOR	=	\$FFFA	; nmi vector location
RESET_VECTOR	=	\$FFFC	; reset vector location
IRQ_VECTOR	=	\$FFFE	; interrupt vector location

.if(0)

VIC II Graphics Chip Definitions and Equates

.endif

```

mob0xpos = vicbase           ;$D000 Sprite x and y positions
mob0ypos = $D001
mob1xpos = $D002
mob1ypos = $D003
mob2xpos = $D004
mob2ypos = $D005
mob3xpos = $D006
mob3ypos = $D007
mob4xpos = $D008
mob4ypos = $D009
mob5xpos = $D00A
mob5ypos = $D00B
mob6xpos = $D00C
mob6ypos = $D00D
mob7xpos = $D00E
mob7ypos = $D00F
msbxpos  = $D010

grcntrl1 = $D011             ;graphics control register, i.e.,
                             ;msb raster/ECM/BMM/DEN/RSEL/y scroll
                             ;bits defined for use with above reg.

st_ecm    = $40
st_bmm    = $20
st_den    = $10
st_25row  = $08
rasreg    = $D012            ;raster register
lp xpos   = $D013            ;light pen x position
lp ypos   = $D014            ;light pen y position
mobenable = $D015            ;moving object enable bits.
grcntrl2  = $D016            ;graphics control register, i.e.,:
                             ;RES/MCM/CSEL/x scroll
                             ;bits defined for use with above reg.

st_mcm    = $10
st_40col  = $08
moby2     = $D017            ;double object size in y
grmemptr  = $D018            ;graphics memory pointer VM13-VM10|CB13-CB11
                             ;i.e., video matrix and character base.

grirq     = $D019            ;graphics chip interrupt register.
grirqen   = $D01A            ;graphics chip interrupt enable register.
st_rasen  = $01              ;bit to enable raster interrupt in grirqen
mobprior  = $D01B            ;moving object to background priority
mobmcm    = $D01C            ;moving object multicolor mode select.
mobx2     = $D01D            ;double object size in x
mobmobcol = $D01E            ;object to object collision register.
mobbakcol = $D01F            ;object to background collision register.

```

```

extclr   = $D020           ;exterior(border) color.
bakclr0  = $D021           ;background #0 color
bakclr1  = $D022           ;background #1 color
bakclr2  = $D023           ;background #2 color
bakclr3  = $D024           ;background #3 color
mcmclr0  = $D025           ;object multicolor mode color 0
mcmclr1  = $D026           ; " " " " " 1
mob0clr  = $D027           ;object color
mob1clr  = $D028           ;object color
mob2clr  = $D029           ;object color
mob3clr  = $D02A           ;object color
mob4clr  = $D02B           ;object color
mob5clr  = $D02C           ;object color
mob6clr  = $D02D           ;object color
mob7clr  = $D02E           ;object color
obj0Pointer = COLOR_MATRIX+$3F8 ;$8FF8pointer to object graphics RAM
obj1Pointer = $8FF9
obj2Pointer = $8FFA
obj3Pointer = $8FFB
obj4Pointer = $8FFC
obj5Pointer = $8FFD
obj6Pointer = $8FFE
obj7Pointer = $8FFF

```

```

.if(0)

```

Sound Interface Device (SID) Chip Definitions and Equates

(6581)

```

.endif

```

```

vlfreqlo = sidbase           ;$D400 write only.
vlfreqhi = $D401             ;write only
vlpwlo   = $D402             ;write only.
vlpwhi   = $D403             ;write only.
v1cntrl  = $D404             ;write only.
v1attdec = $D405             ;write only.
v1susrel = $D406             ;write only.
v2freqlo = $D407             ;write only.
v2freqhi = $D408             ;write only.
v2pwlo   = $D409             ;write only.
v2pwhi   = $D40A             ;write only.
v2cntrl  = $D40B             ;write only.
v2attdec = $D40C             ;write only.
v2susrel = $D40D             ;write only.

```

```

v3freqlo      =      $D40E      ;write only.
v3freqhi      =      $D40F      ;write only.
v3pwlo        =      $D410      ;write only.
v3pwhi        =      $D411      ;write only.
v3cntrl       =      $D412      ;write only.
v3attdec      =      $D413      ;write only.
v3susrel      =      $D414      ;write only.
fclo          =      $D415      ;write only.
fchi          =      $D416      ;write only.
resfilt       =      $D417      ;write only.
modevol       =      $D418      ;write only.
potx          =      $D419      ;read only.
poty          =      $D41A      ;read only.
osc3rand      =      $D41B      ;read only.
env3          =      $D41C      ;read only.

```

```
.if(0)
```

Communications Interface Adapter (CIA) Chip Definitions and Equates (6526)

```
.endif
```

```

cialpra       =      cialbase    ;$DC00 cial peripheral data reg a.
cialprb       =      $DC01      ;cial peripheral data reg b.
cialddra      =      $DC02      ;cial data direction reg a.
cialddrb      =      $DC03      ;cial data direction reg b.
cialtalo      =      $DC04      ;cial timer a low reg.
cialtahi      =      $DC05      ;cial timer a hi reg.
cialtblo      =      $DC06      ;cial timer b lo reg.
cialtbhi      =      $DC07      ;cial timer b hi reg.
cialtod10ths  =      $DC08      ;cial 10ths of sec reg.
cialtodsec    =      $DC09      ;cial seconds reg.
cialtodmin    =      $DC0A      ;cial minutes reg.
cialtodhr     =      $DC0B      ;cial hours - AM/PM reg.
cialskr       =      $DC0C      ;cial serial data reg.
cialicr       =      $DC0D      ;cial interrupt control reg.
cialcra       =      $DC0E      ;cial control reg a.
cialcrb       =      $DC0F      ;cial control reg b.

cia2pra       =      cia2base    ;$DD00 cia2 peripheral data reg a.
cia2prb       =      $DD01      ;cia2 peripheral data reg b.
cia2ddra      =      $DD02      ;cia2 data direction reg a.
cia2ddrb      =      $DD03      ;cia2 data direction reg b.
cia2talo      =      $DD04      ;cia2 timer a low reg.
cia2tahi      =      $DD05      ;cia2 timer a hi reg.

```

```

cia2tblo    =      $DD06      ;cia2 timer b lo reg.
cia2tbhi    =      $DD07      ;cia2 timer b hi reg.
cia2todl0ths =      $DD08      ;cia2 10ths of sec reg.
cia2todsec   =      $DD09      ;cia2 seconds reg.
cia2todmin   =      $DD0A      ;cia2 minutes reg.
cia2todhr    =      $DD0B      ;cia2 hours - AM/PM reg.
cia2sdr      =      $DD0C      ;cia2 serial data reg.
cia2icr      =      $DD0D      ;cia2 interrupt control reg.
cia2cra      =      $DD0E      ;cia2 control reg a.
cia2crb      =      $DD0F      ;cia2 control reg b.

```

```

;
;           Zero Page
;
; 00-01 >> 6510 I/O registers
; 02-33 >> General purpose two byte pseudoregisters named r0-r15.
;           These are used to pass arguments between the application
;           to the GEOS Kernal routines and to store temporary data.
;           These registers are saved at the beginning of
;           interrupt level code enabling sharing of routines that use
;           these registers between main loop code and interrupt level
;           code.
; 34-6F >> OS global variables.
; 70-7F >> Zpage reserved for application, pseudoregisters a2-a9
; 80-FA >> Currently reserved for c64 Kernal usage ONLY!
; FB-FE >> Application general purpose registers named a0-a1. These
;           provided so that applications have a little sacred space of
;           their own. They are provided courtesy of the c64
;           programmers reference guide, which claims that they are "Free
;           zpage space for user programs"
; FF >> Used by Kernal ROM and BASIC

```

```

; Pseudoregisters, described above
;
; .zsect $00
zpage:
; .block 2 ; 6510 registers
r0:    =      $02
r0L    =      $02
r0H    =      $03

r1:    =      $04
r1L    =      $04
r1H    =      $05

```

r2:	=	\$06
r2L	=	\$06
r2H	=	\$07

r3:	=	\$08
r3L	=	\$08
r3H	=	\$09

r4:	=	\$0A
r4L	=	\$0A
r4H	=	\$0B

r5:	=	\$0C
r5L	=	\$0C
r5H	=	\$0D

r6:	=	\$0E
r6L	=	\$0E
r6H	=	\$0F

r7:	=	\$10
r7L	=	\$10
r7H	=	\$11

r8:	=	\$12
r8L	=	\$12
r8H	=	\$13

r9:	=	\$14
r9L	=	\$14
r9H	=	\$15

r10:	=	\$16
r10L	=	\$16
r10H	=	\$17

r11:	=	\$18
r11L	=	\$18
r11H	=	\$19

r12:	=	\$1A
r12L	=	\$1A
r12H	=	\$1B

r13:	=	\$1C
r13L	=	\$1C
r13H	=	\$1D

r14: = \$1E
r14L = \$1E
r14H = \$1F

r15: = \$20
r15L = \$20
r15H = \$21

; Pseudoregister names in all lower case

;
r0l = r0L
r0h = r0H

r1l = r1L
r1h = r1H

r2l = r2L
r2h = r2H

r3l = r3L
r3h = r3H

r4l = r4L
r4h = r4H

r5l = r5L
r5h = r5H

r6l = r6L
r6h = r6H

r7l = r7L
r7h = r7H

r8l = r8L
r8h = r8H

r9l = r9L
r9h = r9H

r10l = r10L
r10h = r10H

r11l = r11L
r11h = r11H

r12l = r12L
r12h = r12H


```

r13l    =    r13L
r13h    =    r13H

```

```

r14l    =    r14L
r14h    =    r14H

```

```

r15l    =    r15L
r15h    =    r15H

```

```

;
;    All variables starting here are saved when
;    DB's and DA's are swapped in.

```

```

s_zp_global:

```

```

;
;    This local variable is for the graphics routines
;
currentPattern:    ; $0022 pointer to the current pattern
    .block 2

```

```

;
;    These local variables are for string input
;
string:            ; $0024
    .block 2

```

```

;
;    Global variables used by the card set manager to define the
;    current variable sized card set. The data structure is explained in
;    the file CardSet_manager.
;

```

```

cardData:
baselineOffset:    ; $0026 Offset from top line to baseline in character set
    .block 1
currentSetWidth:    ; $0027 Card width in pixels
    .block 2
currentHeight:      ; $0029 Card height in pixels
    .block 1
currentIndexTable:  ; $002A Size of each card in bytes
    .block 2
cardDataPointer:    ; $002C Pointer to the actual card graphics data
    .block 2
endCardData:        ; $002C

```

```

CARD_DATA_ITEMS =    endCardData - cardData
;

```

```

;      Current character drawing mode
;
;          %1xxxxxxx > underline
;          %0xxxxxxx > no underline
;          %x1xxxxxx > italic
;          %x0xxxxxx > no italic
;          %xx1xxxxx > reverse video
;          %xx0xxxxx > no reverse video
;
currentMode:      ; $002E Current underline, italic and reverse flags
        .block 1
;
;      This flag indicates whether double buffering is on
;
displayBufferOn:  ; $002F bit 7 controls writes to foreground screen
        .block 1 ;      bit 6 controls writes to background screen
;
;      These global variables support the mouse mode
;
mouseOn:          ; $0030 flag indicating that the mouse mode is on
        .block 1 ;bit 7 - mouse on
                ;bit 6 - menu on
                ;bit 5 - icons on
mousePicture:     ; $0031 pointer to mouse graphics data
        .block 2
;
;      These global variables are for clipping.  Only the character drawing
;      routines support these variables in V1.1
;
windowTop:
        .block 1 ; $0033 top line of window for text clipping
windowBottom:
        .block 1 ; $0034 bottom line of window for text clipping
leftMargin:      ;      leftmost point for writing characters. CR
        .block 2 ; $0035 will return to this point
rightMargin:     ;      rightmost point for writing characters. When
        .block 2 ; $0037 crossed, call mode through StringFaultVector
;
;
;      This is the end of zpage global RAM saved by DA's and DB's.

e_zp_global:
;
;      This flag indicates that a new key has been pressed
;

```

```

pressFlag:          ; $0039 Flag indicating that a new key has been presses
    .block 1        ;bit 7 - key data is new
                    ;bit 6 - disk data is new
                    ;bit 5 - mouse data is new
;
;      Mouse positions
;
mouseXPosition:      ; $003A x position of mouse
    .block 2
mouseYPosition:      ; $003C y position of mouse
    .block 1
;
;      These are variables for the inline calls
;
returnAddress:
    .block 2        ; $003D address to return from inline call

.if(0)

```

Application zpage Space NOT to Be Used by GEOS or DA's

```

.endif

```

```

a0      =      $FB
a0L     =      a0
a0H     =      a0+1
a1      =      $FD
a1L     =      a1
a1H     =      a1+1

```

```

;      alternate names for the user
;      Pseudoregisters.
;

```

```

a2      =      $70
a2L     =      $70
a2H     =      $71

```

```

a3      =      $72
a3L     =      $72

```

```

a3H     =      $73
a3l     =      $72
a3h     =      $73
a3l     =      $72

```

a4 = \$74
a4l = \$74
a4h = \$75
a4L = \$74

a5 = \$76
a5l = \$76
a5h = \$77
a5L = \$76
a5H = \$77

a6 = \$78
a6l = \$78
a6h = \$79
a6L = \$78
a6H = \$79

a7 = \$7A
a7l = \$7A
a7h = \$7B
a7L = \$7A
a7H = \$7B

a8 = \$7C
a8l = \$7C
a8h = \$7D
a8L = \$7C
a8H = \$7B

a9 = \$7E
a9l = \$7E
a9h = \$7F
a9L = \$7E
a9H = \$7F

.if(0)

Start of GEOS System RAM

.endif

;
;
; NOTE that this area starts with the 256 byte disk
; buffers, which ought to be kept on 256 byte boundaries.

.ramsect OS_VARS ;\$8000

```

diskBlkBuf:      ; $8000 general disk block buffer
                .block 256
fileHeader:      ; $8100 block used to hold the header block for a GEOS file
                .block 256
curDirHead:      ; $8200 block contains directory header information for
                .block 256
                ; $8300 disk in currently selected drive.

fileTrScTab:     ; $8400 buffer used to hold track and sector chain for
                .block 256 ; a file of maximum size 32,258 bytes.
                ; In order to save a file larger than this,
                ; use the partial file read/write routines,
                ; and a special T,S table of sufficient size.

dirEntryBuf:     ; $8500 buffer used to build a files directory entry
                .block 30

```

```
.if(0)
```

Disk Variables

```

.endif

;      These RAM arrays hold the name of the current disk
;      in each of the two possible drives.
;
DrACurDkNm:      ; $841E Disk name of disk in drive A
                .block 18 ; 16 char disk name, 2 char ID
DrBCurDkNm:      ; $8430 Disk name of disk in drive B
                .block 18 ; 16 char disk name, 2 char ID

dataFileName:    ; $8442 Name of data file (passed to application)
                .block 17
dataDiskName:    ; $8453 Disk that data file is on.
                .block 18

PrntFilename:    ; $8465 Name of current printer driver
                .block 17 ; 16 byte filename, 1 byte terminator
PrntDiskName:    ; $8476 Disk that current printer driver resides on
                .block DK_NM_ID_LEN+1 ; disk name plus terminator byte

curDrive:        ; $8489 currently active disk drive (8,9,10 or 11)
                .block 1
diskOpenFlg:     ; $848A indicates if a disk is currently open
                .block 1

```

```

isGEOS:                ; $848B flag indicates if current disk is a GEOS disk
    .block 1

interleave:
    .block 1            ; $848C BlkAlloc uses the value here as the desired
                        ; interleave when selecting
                        ; free blocks for a disk chain.  GEOS
                        ; initializes this value to 8.

numDrives:              ; $848D # of drives running on system.
    .block 1

driveType:              ; $848E Disk Drive types -- not yet used.
    .block 4            ; 1 byte drive type for each of drives
                        ; 8,9,10,11
                        ; Values in these variables can
                        ; be found in Constants, and appear like
                        ; "DRV_1541"

turboFlags:             ; $8492 Turbo state flags -- only the first two are
                        ; presently used (2 drive system)
    .block 4            ; Turbo flag for each of drives 8,9,10 and 11

;      Variables kept current for a specific opened file of
;      structure type VLIR (Variable Length Indexed Record)

curRecord:
    .block 1            ; $8496 current record #

usedRecords:
    .block 1            ; $8497 number of records in open file

fileWritten:
    .block 1            ; $8498 flag indicating if file has been
                        ; written to since last update
                        ; of index Tab & BAM

fileSize:
    .block 2            ; $8499 current size (in blocks) of file
                        ; This is pulled in from &
                        ; written to directory entry.

```

The following variables are saved for DB's and DA'S

```
s_nonzp_global:
```

```
.if(0)
```

Vectors

```
.endif
```

```
applicationMain:      ; $849C Application's main loop code. Allows
.block 2              ; applications to include their own main loop at the
                      ; end of OS main loop
interruptTopVector:   ; $849D Called at the top of OS interrupt code to allow
.block 2              ; application programs to have interrupt level routines
interruptBottomVector:
                      ; $849F Called at bottom of OS interrupt code to allow
.block 2              ; application programs to have interrupt level routines
                      ; mouseVector: $84A1 routine to call on mouse key press
.block 2
keyVector:            ; $84A3 routine to call on keypress
.block 2
inputVector:          ; $84A5 routine to call on input device change
.block 2
mouseFaultVector:     ; $84A7 routine to call when mouse goes outside region
.block 2              ; or off a menu
otherPressVector:     ; $84A9 routine to call on mouse press that is not a
                      ; menu
.block 2              ; or an icon
StringFaultVector:    ; $84AB vector for when character written over
.block 2              ; rightMargin
alarmTmtVector:       ; $84AD address of a service routine for the alarm
.block 2              ; clock time-out (ringing, graphic etc.) that
                      ; the Application can use if necessary. Normally 0.
BRKVector:            ; $84AF routine called when BRK encountered
.block 2
RecoverVector:        ; $84B1 routine called to recover background behind
.block 2              ; menus and dialog boxes

;
;      This variable determines the speed at which
;      menu items and icons are flashed.

selectionFlash:       ; $84B3
.block 1

;
;      This global variable is for string input
;
alphaFlag:            ; $84B4 flag for alphanumeric input
.block 1
```

;This RAM variable contains flag bits in b7 and b6 to specify how the system should indicate icon selection to the user. If no bits are set, then the system does nothing to indicate icon selection, and the service routine is simply called.

; The possible flags are: ST_FLASH = \$80 ; flash the icon
 ST_INVERT = \$40 ; invert the selected icon
 ;If ST_FLASH is set, the ST_INVERT flag is ignored and the icon flashes
 ;but is not inverted when the programmer's routine is called. If ST_INVERT
 ;is set, and ST_FLASH is CLEAR, then the icon will be inverted when the
 ;programmer's routine is called.

```
iconSelFlag:          ; $84B5
    .block 1
```

```
;
;       This variable holds the current mouse faults
;
```

```
faultData:           ; $84B6 Bit flags for mouse faults
    .block 1
```

```
.if(0)
```

MouseRelated Global Variables

```
.endif
```

```
menuNumber:          ; $84B7 number of currently working menu
    .block 1
```

```
mouseTop:            ; $84B8 top most position for mouse
    .block 1
```

```
mouseBottom:         ; $84B9 bottom most position for mouse
    .block 1
```

```
mouseLeft:           ; $84BA left most position for mouse
    .block 2
```

```
mouseRight:          ; $84BC right most position for mouse
    .block 2
```

```
;
;       Global variables for string input and prompt manipulation
;
```

```
stringX:             ; $84BE X position for string input
    .block 2
```

```
stringY:             ; $84C0 Y position for string input
    .block 1
```

```
e_nonzp_global:      ; $84C1
```

```
;
```



```

;      This is the end of non-zpage global RAM
;      saved by DA's and DB's

;
mousePicData:
    .block 64      ; $84C1 RAM array for mouse picture data.
maximumMouseSpeed:  ; $8501 maximum speed for mouse
    .block 1
minimumMouseSpeed:  ; $8502 minimum speed for mouse
    .block 1
mouseAcceleration:  ; $8503 acceleration of mouse
    .block 1

;
;      These global variables hold the state of
;      the keyboard and the mouse
;
keyData:            ; $8504 This is where key service routines should look
    .block 1
mouseData:          ; $8505 This is where mouse service routines should look
    .block 1
inputData:          ; $8506 This is where input drivers pass device specific
    .block 4      ;      information to applications that want it

;
;      Random number, incremented each interrupt
;
random:
    .block 2      ; $850A

saveFontTab:
    .block 9      ; $850C when going into menus, save user active font
                  ;      table here

;The following RAM variable is used to determine double clicks on icons.
;dblClickCount is loaded with a value when an icon is first "clicked on".
;It is decremented each interrupt,
;if it is nonzero when the icon is again selected, then the double click
;flag (r0H) is passed to the service routine with a value of TRUE. If the
;dblClickCount variable is zero when the icon is clicked on, then the flag
;is passed with a value of FALSE

dblClickCount:      ; $8515 used to determine double clicks on icons.
    .block 1

;      RAM variables associated with time of day clock

year: .block 1      ; $8516
month: .block 1     ; $8517
day: .block 1       ; $8518
hour: .block 1      ; $8519
minutes: .block 1   ; $851A
seconds: .block 1   ; $851B
;

```

```

;      This global variable is for alarm support
;
alarmSetFlag:
    .block 1      ; $851C is 0 if the alarm is not set for geos to
                  ;      monitor,$FF if set.
;
;      These RAM variables are associated with dialog handling
;

sysDBData:      ;$851D
    .block 1      ;used internally to indicate which command caused
                  ;a return to the application (in dialog boxes)
                  ;actual data returned in r0L.
screenColors:    ;$851E default screen colors
    .block 1

;
;      The following equate is used to define the size of the buffer needed
;to save away all of the system RAM variables necessary to specify the state
;of an application.
;      This equate corresponds to all of the RAM variables specified in the
;SaveSysRamTab in the SaveSysRam module
;


---


MENU_SPACE      =      (3 * MAXIMUM_MENU_NESTING) + (2 * MAXIMUM_MENU_ITEMS)
                  =      $002A
SRAM_ZP_GLOBAL
                  =      e_zp_global - s_zp_global
                  =      $0017
SRAM_NONZP_GLOBAL
                  =      e_nonzp_global - s_nonzp_global
                  =      $0026
SRAM_LOCAL
                  =      2 + 14 + MENU_SPACE + (MAXIMUM_PROCESSES*8)
                  +      (SLEEP_MAXIMUM*4)
                  =      $012A
SRAM_SPRITES
                  =      $0026

TOT_SRAM_SAVED
                  =      SRAM_ZP_GLOBAL + SRAM_NONZP_GLOBAL + SRAM_LOCAL
                  +      SRAM_SPRITES + 20
                  =      $01A1

dlgBoxRamBuf:
                  =      $01A1

    .block TOT_SRAM_SAVED ;sum of locations specified in SaveSysRamTab

.page
.end

```

Appendix C.

Routines

This file contains the equates for the GEOS jump table.

```
;      MISC
;      The following two jump table entries are the only
;      ones at $C000.  The rest are at OS_JUMPTAB;

BootGEOS      =      $C000      ; Re-Boot all of GEOS from only
                                ; $C000 to $C02F preserved.
ResetHandle   =      $C003      ; Enter GEOS kernal as if COLD start


InterruptMain =      C100

;
;      PROCESSES
;
InitProcesses =      C103
RestartProcess =      C106
EnableProcess =      C109
BlockProcess  =      C10C
UnblockProcess =      C10F
FreezeProcess =      C112
UnfreezeProcess =      C115
;
```

GRAPHICS

```
HorizontalLine = C118
InvertLine     = C11B
RecoverLine    = C11E
VerticalLine   = C121
Rectangle      = C124
FrameRectangle = C127
InvertRectangle = C12A
RecoverRectangle = C12D
DrawLine       = C130
DrawPoint      = C133
GraphicsString = C136
SetPattern     = C139
GetScanLine    = C13C
TestPoint      = C13F
```

BACKGROUND GENERATION

```
BitmapUp = C142
```

CHARACTER MANIPULATION

```
PutChar      = C145
PutString    = C148
UseSystemFont = C14B
```

MOUSE & MENUS

```
StartMouseMode = C14E
DoMenu         = C151
RecoverMenu    = C154
RecoverAllMenus = C157
DoIcons        = C15A
```

UTILITIES

```
DShiftLeft = C15D
BBMult      = C160
BMult       = C163
DMult       = C166
Ddiv        = C169
DSdiv       = C16C
Dabs        = C16F
Dnegate     = C172
Ddec        = C175
ClearRam    = C178
FillRam     = C17B
MoveData    = C17E
InitRam     = C181
PutDecimal  = C184
GetRandom   = C187
```

;
;
;
MISC MOUSE, MENU, GRAPHICS, SLEEP

MouseUp = C18A
 MouseOff = C18D
 DoPreviousMenu = C190
 ReDoMenu = C193
 GetSerialNumber = C196
 Sleep = C199
 ClearMouseMode = C19C
 i_Rectangle = C19F
 i_FrameRectangle = C1A2
 i_RecoverRectangle = C1A5
 i_GraphicsString = C1A8

;
;
;
BACKGROUND GENERATION

i_BitmapUp = C1AB

;
;
;
CHARACTER MANIPULATION

i_PutString = C1AE
 GetRealSize = C1B1

;
;
;
MOUSE & MENUS

;
;
;
UTILITIES

i_FillRam = C1B4
 i_MoveData = C1B7

;
;
;
ROUTINES ADDED LATER

GetString = C1BA

 GotoFirstMenu = C1BD
 InitTextPrompt = C1C0
 MainLoop = C1C3
 DrawSprite = C1C6
 GetCharWidth = C1C9
 LoadCharSet = C1CC
 PosSprite = C1CF
 EnablSprite = C1D2
 DisablSprite = C1D5
 CallRoutine = D1D8
 CalcBlksFree = C1DB

ChkDkGEOS	=	C1DE
NewDisk	=	C1E1
GetBlock	=	C1E4
PutBlock	=	C1E7
SetGEOSDisk	=	C1EA
SaveFile	=	C1ED
SetGDirEntry	=	C1F0
BldGDirEntry	=	C1F3
GetFreeDirBlk	=	C1F6
WriteFile	=	C1F9
BlkAlloc	=	C1FC
ReadFile	=	C1FF
SmallPutChar	=	C202
FollowChain	=	C205
GetFile	=	C208
FindFile	=	C20B
CRC	=	C20E
LdFile	=	C211
EnterTurbo	=	C214
LdDeskAcc	=	C217
ReadBlock	=	C21A
LdApplic	=	C21D
WriteBlock	=	C220
VerWriteBlock	=	C223
FreeFile	=	C226
GetFHdrInfo	=	C229
EnterDeskTop	=	C22C
StartAppl	=	C22F
ExitTurbo	=	C232
PurgeTurbo	=	C235
DeleteFile	=	C238
FindFTypes	=	C23B
RstrAppl	=	C23E
ToBasic	=	C241
FastDelFile	=	C244
GetDirHead	=	C247
PutDirHead	=	C24A
NxtBlkAlloc	=	C24D
ImprintRectangle	=	C250
i_ImprintRectangle	=	C253
DoDlgBox	=	C256
RenameFile	=	C259
InitForIO	=	C25C
DoneWithIO	=	C25C
DShiftRight	=	C262
CopyString	=	C255
CopyFString	=	C268
CmpString	=	C26B
CmpFString	=	C26E
FirstInit	=	C271
OpenRecordFile	=	C274
CloseRecordFile	=	C277
NextRecord	=	C27A

PreviousRecord	=	C27D
PointRecord	=	C280
DeleteRecord	=	C283
InsertRecord	=	C286
AppendRecord	=	C289
ReadRecord	=	C28C
WriteRecord	=	C28F
SetNextFree	=	C292
UpdateRecordFile	=	C295
GetPtrCurDkNm	=	C298
PromptOn	=	C29B
PromptOff	=	C29E
OpenDisk	=	C2A1
DoInlineReturn	=	C2A4
GetNextChar	=	C2A7
BitmapClip	=	C2AA
FindBAMBit	=	C2AD
SetDevice	=	C2B0
IsMseInRegion	=	C2B3
ReadByte	=	C2B6
FreeBlock	=	C2B9
ChangeDiskDevice	=	C2BC
RstrFrmDialog	=	C2BF
Panic	=	C2C2
BitOtherClip	=	C2C5
StashRAM	=	C2C8
FetchRAM	=	C2CB
SwapRAM	=	C2CE
VerifyRAM	=	C2D1
DoRAMOp	=	C2D4
*		



Appendix D.

Data File Formats

Overview

This chapter describes the output file data formats of the Text Scrap, Photo Scrap, and geoWrite files. The Photo Scrap and Text Scrap files are designed so that text and graphics data can be shared between applications. This is the format used by the Photo and Text manager desk accessories. Both the Text and Photo Scraps are stored as sequential system files on disk. When the user performs a cut or copy operation from inside an application, a Photo Scrap or Text Scrap file is created on the application disk. The user can then quit the present application, start up a new one and paste the contents of the Scrap file into the new application's document. Scraps can also be collected into Albums using the Photo Manager or Text Manager desk accessories. The geoWrite output format is important for programmers desiring to output geoWrite format from their programs or read geoWrite documents into their documents.

Section two of this paper describes the Photo Scrap. Section three documents the Text Scrap. The final section describes geoWrite's format.

Future Releases

The Photo and Text Scrap formats have been expanded in the past to include new features and may be expanded in the future. To avoid problems, applications should check the version string in the File Header block of the Text or Photo Scrap files before using the data. Checking version strings is described in the File System chapter. Bytes 89 - 92 (decimal) of the File Header contain the ASCII string, V1.1, or a later version of it. Version 1.1 was the first general release format contained in any data file.

If the scrap file is an older format than your application supports, it will have to be converted, something the application will probably want to provide. If the format is newer than the application, then the application should refrain from using it.

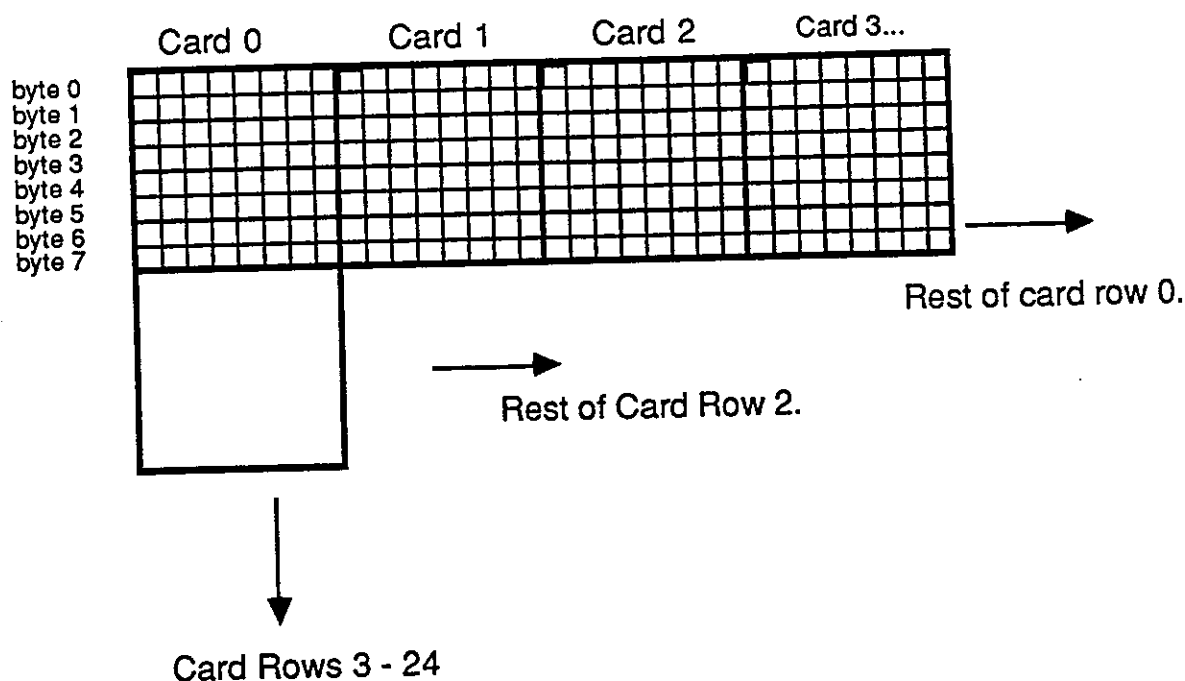
Photo Scrap

The Photo Scrap presently supports a single Bit-Mapped Object. A Bit-Mapped Object is a GEOS object for storing compacted bit-mapped data. Compacted data and Bit-Mapped Objects are described in detail in the Graphics chapter. Photo Scraps consist of a Bit-Mapped Object followed by compacted Color Table for the bit-mapped area described by the Bit-Mapped Object.

In uncompact form, the Color Table contains one byte of color information for each card generated by uncompacting the Bit-Mapped Object. The Color Table bytes are taken from the one thousand byte color table that normally determines the colors of each of the cards on the the screen in standard high-resolution bit-mapped mode. A card, as referred to here, is the same as a Programmable Character as described in the c64 manual. The reader is referred to the description of bit-mapped graphics, cards, and color bytes starting on page 121 of Commodore 64 Programmer's Reference Guide.

In c64 hi-res bit-mapped mode, a card takes up eight bytes and defines an eight pixel wide by eight pixel high square on the screen. Each card is associated with a byte which determines its color. For example, the first color byte in the Color Table controls the color of the upper left most card on the screen. The second color byte determines the color of the second 8 x 8 card which appears just to the right of the first card and so on.

A diagram of the organization of bytes in the bit-mapped mode screen is:



Byte Organization in Bit-Map Screen

Photo Scraps are not limited to the size of the screen. While most applications create scraps which are smaller than the full screen, there will eventually be those which will construct a Scrap from an object larger than the screen size. The Color Table and bit-mapped data may be greater or less than full screen size for hi-res bit-mapped mode.

Consequently, three bytes containing the dimensions of the Bit-Mapped Object appear before the first COUNT/Bit-map pair. The first byte contains the width of the bitmap in bytes and is followed by two bytes containing the height in scanlines. Multiplying the two together gives the total number of graphic bytes to be generated by the following COUNT/Bit-map pairs. The height must always be divisible by 8 as only complete card rows are cut or copied to the Photo Scrap. The width of the scrap is always in complete cards. These restrictions are necessary because each color byte represents the color of a complete 8 byte card.

The color table is compacted using the same compaction schemes used to compact the Bitmap Object into COUNT/Bit-map pairs. Thus even the color information appears in the PhotoScrap as a series of COUNT/Bit-mapped pairs. The Color Table COUNT/Bit-map

starts just after the last graphics COUNT/Bit-map. After the proper number of graphics data bytes have been uncompacted, the next COUNT/Bit-map pair begins the compacted Color Table. The number of data bytes divided by 8 gives you the number of ColorTable bytes to be uncompacted. The Figure below shows the structure of the Photo Scrap.

Photo Scrap Data Format		
Byte Number	Contents	Purpose
0	Width	The width in bytes of the bitmap picture
1-2	Height	The height in scanline of the bitmap picture
3	Count	Three modes for storing bitmap data depending on Count: 0-127: use next byte COUNT times (repeat count) 128-220: use next (COUNT-128) bytes once each (straight bitmap) 221-255: use next byte as BIGCOUNT (a repeat count), repeat the following (COUNT -220) bytes BIGCOUNT times
4-end of bitmap	Bitmap Data	The bitmap data in one of the three COUNT modes
--	Count	New mode byte
-	Bitmap Data	The bitmap data in one of the three COUNT modes
-		More Count/Bitmap Pairs
-	Color Table	Color Table stored compacted. One color byte generated for each uncompacted card.

To summarize, the Photo Scrap is made up of three-dimension bytes, followed by one large compacted Bit-Mapped Object, followed by a Color Table. Both the Bit-Mapped Object and the Color Table are a collection of COUNT/Bit-map pairs in different compaction formats. A COUNT/Bit-map pair consists of a format byte followed by a series of data bytes in the indicated compaction format. As described in the graphics chapter in this manual, uncompacted Bit-Mapped Object data must be reordered from scanlines to cards. The Color

Table contains, in compacted form the Bit-Mapped Mode color bytes for each 8 by 8 card defined by the uncompactd Bit-Mapped Object.

Text Scrap V1.2

This section describes the V1.2 Text Scrap. The V2.0 Text Scrap is a superset of the V1.2 Text Scrap. The only addition to Text Scraps for V2.0 is a ruler escape that contains positioning information. The ruler escape and is described in the next section.

The Text Scrap is an ASCII string with embedded escape characters. The escape characters are requisitioned from the nonprintable ASCII chars, sometimes called control chars*. There are two escape chars found in Text Scraps. First is TAB (char \$9). It is up to the application to support or not to support tabs as it wishes. The second escape character is given the constant name NEWCARDSET (\$23). It signals the beginning of a 4 byte font/style escape string. The first two bytes after NEWCARDSET are the font ID of the font to be used to display the following text. The final byte in the string indicates the style of the following text: plain, bold, italic underline and/or outline. Each style is controlled by a bit in the style byte. Setting the bold bit, for example turns on bold face. The significance of each bit is shown below.

A complete NEWCARDSET escape string will appear whenever there is a change in either font or style. The Text Manager desk accessory will not display tabs, font and style changes but they are stored within the Text Scrap nonetheless. Applications must expect these special characters, in addition to regular ASCII characters within the Text Scrap file. The structure of the Text Scrap is shown immediately below.

Text Scrap

The Text Scrap file, as it appears in memory, begins with two bytes which contain the total number of bytes to follow. (Note that these bytes don't count themselves in the total.) After these two count bytes follows a mandatory NEWCARDSET escape string.

* In ASCII the normal printable character set starts with the character '0' which has a number \$20. The first \$20ASCII characters (\$0 - \$1F), are unprintable as they don't correspond to any letter or number like 'a' or '0'. These characters are often used to embed command strings in text.

The escape string is four bytes long and begins with NEWCARDSET. The next two bytes are the font ID number. The low 6 bits of this word contain the point size of the font. The upper 10 bits contain a unique number for the font. The font word is followed by a style byte in which each bit signifies a style, as shown in the table above. Setting a bit in the style byte will turn its associated function on. Clearing the bit turns the function off. All style bits reset to 0 indicates plain text printing.

Text Scrap File Format				
Byte number	Contents	Purpose		
0-1	Length	Number of bytes to follow in file		
2	NEWCARDSET	NEWCARDSET (= \$17). Start of Font/Style command string.		
3-4	Font ID	The low 6 bits of font ID is the point size of the font. The upper 10 bits is the unique number of the font in which the following text should appear.		
5	Style Byte	Constant	Value	Function
		SET_UNDERLINE	10000000	Bit 7=1: turn on underlining underlined
		SET_BOLD	01000000	Bit 6=1: turn on bold face
		SET_REVERSE	00100000	Bit 5=1: turn on reverse video
		SET_ITALIC	00010000	Bit 4=1: turn on italics
		SET_OUTLINE	00001000	Bit 3=1: turn on outline
		SET_SUPERSCRIPT	00000100	Bit 2=1: turn on outline
		SET_SUBSCRIPT	00000010	Bit 1=1: turn on outline
		SET_PLAINTEXT	00000000	All bits=0, indicates plain text
6 to end	Text String	The ascii text with embedded tabs, font/style, and if V2.0, ruler escapes.		

GEOS Fonts			
Font Name	Number Top 10 bits	Point Sizes Last 6 Bits	ID Word
BSW	0	9	\$00 09
University	1	6	\$00 46
		10	\$00 4A
		12	\$00 4C
		14	\$00 4E
		18	\$00 52
		24	\$00 58
California	2	10	\$00 8A
		12	\$00 8C
		13	\$00 8D
		14	\$00 8E
		18	\$00 92
Roma	3	9	\$00 C9
		12	\$00 CC
		18	\$00 D2
		24	\$00 D6
Dwinelle	4	18	\$01 12
Cory	5	12	\$01 4C
		13	\$01 4D

geoLaser Fonts			
Font Name	Number Top 10 bits	Point Sizes Last 6 Bits	ID Word
Commodore	26	10	\$06 8A
LW_Roma	27	9	\$06 C9
		10	\$06 CA
		12	\$06 CC
		14	\$06 CE
		18	\$06 D2
		24	\$06 D8
LW_Cal	28	9	\$07 09
		10	\$07 0A
		12	\$07 0C
		14	\$07 0E
		18	\$07 12
		24	\$07 18
LW_Greek	29	9	\$07 49
		10	\$07 4A
		12	\$07 4C
		14	\$07 4E
		18	\$07 52
		24	\$07 58
LW_Barrows	30	9	\$07 89
		10	\$07 8A
		12	\$07 8C
		14	\$07 8E
		18	\$07 92
		24	\$07 98

Fonts 5 - 25 are on GEOS Font Pack 1			
Font Name	Number Top 10 bits	Point Sizes Last 6 Bits	ID Word
Tolman	6	12	\$01 8C
		24	\$01 98
Bubble	7	24	\$01 D8
FontKnox	8	24	\$02 18
Harmon	9	10	\$02 4A
		20	\$02 54
Mykonos	10	12	\$02 8C
		24	\$02 98
Boalt	11	12	\$02 CC
		24	\$02 D8
Stadium	12	12	\$02 30
Tilden	13	12	\$03 0C
		24	\$03 4C
Evans	14	18	\$03 92
Durrant	15	10	\$03 CA
		12	\$03 CC
		18	\$03 D2
		24	\$03 D8
Telegraph	16	18	\$04 12
Superb	17	24	\$04 58
Bowditch	18	12	\$04 8C
		24	\$04 98
Ormand	19	12	\$04 CC
		24	\$04 D8
Elmwood	20	18	\$05 12
		36	\$05 24
Hearst	21	10	\$05 4A
		12	\$05 4C
		18	\$05 52
		24	\$05 58
Brennens	22	18	\$05 92
Channing	23	14	\$05 CE
		16	\$05 D0
		24	\$05 D8
Putnam	24	12	\$06 0C
		24	\$06 18
LeConte	25	12	\$06 4C
		18	\$06 52

The remainder of the string is composed of text with embedded tabs and possibly more NEWCARDSET escape strings. There is no special character appearing as the last character in the scrap so the application must compare the number of bytes read with a total as computed from the first two bytes of the file.

The table on the next page contains the presently supported GEOS fonts. The geoLaser fonts are designed to look as closely as possible to the fonts inside an Apple LaserWriter.®® When preparing documents to be laser writed, these fonts should be used.

To summarize, the Text Scrap begins with a length word, followed by a mandatory Font/Style change command string, and followed by ASCII chars, tabs, and possibly more Font/Style change strings. This is the V1.2 text scrap.

Version 2.0 Ruler Escape

A ruler escape was added to the V2.0 Text Scrap to maintain compatibility with geoWrite files when justification and multiple "rulers" (formatting changes) within the page were added. A ruler escape need not appear anywhere in the text scrap, but if it appears, it will appear at the beginning of the file, or at the beginning of a paragraph. Paragraphs are defined as ending with a CR, so a ruler escape will always be preceeded by a CR. Ruler escapes are 27 bytes long. They contain information about the document's margins, paragraph justification, and color, if supported.

Tabs are not displayed in the Text Manager even though they appear in the ruler data in the file. In applications that use tabs, the tab character causes spacing to the position of the next tab is set. A wrap to the beginning of the next line is done if no tab is defined in the currently active ruler to the right of the position of the embedded tab character. The format of the V2.0 ruler escape is shown below.

Format of Ruler Escape		
Byte	Content	Description
1	ESC_RULER	ESC_RULER constant
2 - 3	Left Margin	Left Margin in pixel positions. Range 0 - 319
4 - 5	Right Margin	Left Margin in pixel positions. Range: Left Margin > Right Margin <=319
6 - 21	Tabs	Each tab is one word long Bit 15 1 for decimal tab, decimal points aligned 0 for normal text tab. bits 14 - 0 bits 0 - 14 are for the tab position, < Right Margin
22 - 23	Paragraph Margin	How far to indent paragraphs. Range is 0 - 319
24	Justification	Bits for justification and line spacing Bits 0 - 1 0 for left justified text 1 for centered text 2 for right justified text 3 for left and right (fully) justified text Bits 3 - 2 0 for single spaced text 1 for one and a half spaced text 2 for double spaced text
25	Text Color	The color of the text. Currently no GEOS application use this byte
26 - 27	Reserved	Reserved for future use

geoWrite Output File Formats

Like the Text Scrap, there is a V1.1 and a V2.0 geoWrite output format. The version numbers are different for the output file formats and the program releases. You will find geoWrite with version strings of V1.2, V1.3, and V2.0 for the Writer's Workshop, while the output file formats are either V1.1 or V2.0.

In both formats, documents are stored in VLIR files. In general, each record in the VLIR file stores one page of text. Some records are used to store pictures and, in the case of V2.0 files, header and footer information. This arrangement is show below.

VLIR Format for geoWrite Files		
Record #	V1.1 Format Files	V2.0 Format Files
0-60	Text Pages	Text Pages
61	Text Page	Header, empty for none
62	Text Page	Footer, empty for none
63	Text Page	Reserved
64-127	Pictures in BitmapUp format	Pictures in BitmapUp format

The major difference between the V1.1 and V2.0 formats is that the Writer's Workshop V2.0 version supports headers and footers. Pages 61-63 may be used to store text pages with the earlier releases of geoWrite, but these will not be carried when editing with the geoWrite V2.0. This is probably not a problem since no one has ever gotten close to actually being able to store a 64 page document on a 1541 disk. When double sided support for the 1571 becomes available this may become possible.

In geoWrite, each document is broken up into separate pages and each page stored in its own VLIR record. A page consists of ruler information followed by text. For a V1.1 geoWrite file the ruler data consists of right and left margin and tab data.

Format of geoWrite V1.1 page		
Byte	Content	Description
0-1	Left Margin	Pixel position of left margin.
2-3	Right Margin	Pixel position of right margin.
4-19	Tabs	Array of 8 words storing pixel positions of tabs .
20-23	NEWCARDSET	Font and Style of page.
24-?	Text and Graphics	Text of document, may contain ruler, font/style, graphics, or page break escapes. PAGE_BREAK = 1, one byte. Causes geoWrite to begin a new page. ESC_GRAPHICS = 16, includes a picture.
Last Byte	EOF = 1	End of file marker.

The text that follows is stored as ASCII. Escape strings are used for font/style changes and for including pictures. The data for each picture is stored in a separate record. All nonempty pages must start with a font/style escape. A font/style escape cannot be followed immediately by another font/style escape. geoWrite files may also include pictures with an ESC_GRAPHICS. The data for the picture is stored in its own record as a Bit-Mapped Object. See the graphics section for the format of a Bit-Mapped Object.

Graphics Escape String		
Byte	Function	Description
0	ESC_GRAPHICS	The escape to graphics control character = 16
1	Width	Picture's width in cards
2-3	Height	Picture's height in scanlines
5	Record Number	Number of the record containing the picture data The picture data is a photo scrap.

geoWrite V2.0

Version 2.0 is similar to V1.2 but includes a more extensive ruler escape. This is the same format as found in Text Scrap files. The file format for V2.0 data files is as follows.

geoWrite V2.0 Page Format	
Byte	Description
0 - 27	Ruler escape string
28 - 31	NEWCARDSET = 23 font/style escape
32 - ...	Text of document, may contain ruler, font/style, graphics, or page break escapes. PAGE_BREAK = 1, one byte. Causes geoWrite to begin a new page. ESC_GRAPHICS = 16, includes a picture
Last byte	EOF = 0 appears as last byte of document.

Further information is also stored in the File Header of V2.0 files. This information includes the height of the footer and header, the page height the document was formatted with (different depending on the selected printer driver), and flags for NLQ and title page modes.

geoWrite V2.0 File Header Information		
Byte	Contents	Description
137-138	Page Number	Page number to print on first page of this file, need not be 1.
139	Title and NLQ	Bit 7 set = make title page (no header, footer on first page). Bit 6 set means turn NLQ fixed width spacing on.
140-141	Header Height	The height in pixels reserved on each page for the header.
142-143	Footer Height	The height in pixels reserved on each page for the footer
144-145	Page Height	Different printers support different vertical resolutions. If the height of the page as stored here does not match what the printer is capable of, then geoWrite 2.0 reformats the file to match the printer.

geoWrite Summary

geoWrite files are divided into pages stored in different records of a VLIR file. These records may also contain bitmap data for pictures included in the document. In addition the V2.0 format includes header, footer, and page height as well as justification, NLQ and title page flags. In V1.1 files, there is only one small ruler – at the top of the page. A different ruler may control each paragraph in V2.0 files.

The above information should be sufficient to enable programmers to read and to create files in any of the formats. It is important to note that each of the earlier versions of output file formats are subsets of the later versions. Thus the V1.1 Text Scrap is a subset of the V2.0 and can be read by the later version Text Manager. The only possible incompatibility between formats is the ability of V1.1 geoWrite to store text pages in the header, footer, and reserved records. As mentioned above, it is unlikely that a 64 page document will fit on one disk.

Text Scraps and geoWrite files differ in that Text Scraps are meant to be only one page or less. The Text Scrap is designed to be a more generic object, enabling a common ground between word processors.

A

APA (All Points Addressable)
 Graphics Mode, 327
 Application code, location of, 15
 Application Main, 12
 ASCII printing, 331, 333-334
 dot-matrix printers, 326-327, 328
 PrintASCII, 334, 341, 352, 373-374
 StartASCII, 331-333, 340, 350, 373
 Assembler directives, 21

B

Background buffer, for screen data,
 14, 68-69
 Bank switching, 19-20
 BBMult, 190
 BetRollBuffer, commodore driver,
 364, 377
 BIGCOUNT, 90
 Bit-Image Mode, 327
 Bit-mapped graphics, 89-98
 BIGCOUNT, 90
 BitmapClip, 93-95
 BitmapUp, 92
 BitOtherClip, 96-98
 Count/bit-map pairs, 89-91, 96
 Countbyte, 89-91
 BitmapUp, 92
 BldGDirEntry, 300
 BlkAlloc, 291-292
 Block Availability Map, 235, 236
 Block Distribution by Track, 234
 Blocked process, 178
 BlockProcess, 182
 BMult, 191
 Boldface, 127
 Buffers, background, for screen data,
 14, 68-69
 Bytes, GEOS, kernal version bytes,
 17-18

C

CalcBlocksFree, 270
 CallRoutine, 210
 CARDSWIDE/CARDSDEEP, 333
 Change Departments, 31, 61-62, 63, 64
 ChangeDepartMenu, 31-32
 ChangeDiskDevice, 215
 Character level routines
 clipping in, 115
 fonts, 128-129, 131
 GetCharWidth, 126
 GetNextChar, 114, 115, 119
 GetRealSize, 118, 125
 InitTestPrompt, 116, 120
 keyData, 115, 127, 1214
 OurGetString, 114, 116, 117-118, 127
 OurStringFault, 118
 PromptOff, 122
 PromptOn, 116, 121
 Putchar, 115, 123-124, 127
 SmallPutChar, 124

StringFaultVector, 115, 124
 style control, 127-128
 as text support, 113
 text wrap, 115
 Characters, points, 128
 Character sets, LoadCharSets,
 130, 131, 132
 ChkDkGEOS, 256
 ClearMouseMode, 148
 Clipping, 115
 CloseRecordFile, 319
 Color, graphics, 68
 Commodore driver
 BetRollBuffer, 364, 377
 FormFeed, 384-385
 GetDimensions, 372
 Greturn, 384
 PrintASCII, 373-374
 PrintBuffer, 369-370
 print driver to disk stuff, 367
 printer equates, 365
 PrintPrintBuffer, 375-376
 RAM storage/utilities, 366
 resident jump table, 366
 Roll8bytesIn, 380
 Roll8bytesOut, 381
 RollaCard, 378
 Rotate, 385
 SetGraphics, 382
 StartASCII, 373
 StartPrint, 368-369
 StopPrint, 371
 TestBuffer, 379
 TopRollBuffer, 363, 364, 376
 UnSetGraphics, 382
 WarmStart Configuration, 386-390
 Commodore Soft-Ware Showcase,
 31, 32, 62-64
 Communications interface adaptor,
 chip, definitions and equates, 415-416
 Compaction formats, 89-91
 Constants, 243
 desk accessory save foreground bit,
 407
 dialog boxes, 403-406
 directory entry, 399-400
 directory header, 399
 disk, 401
 disk error, 402-403
 file, 40
 file header, 400
 file types, 397-398
 flags, 397
 GetFile Equates, 401
 graphics, 395
 keyboard, 394
 menu, 392, 396
 for misc, 391
 mouse, 394
 processes, 392
 standard commodore file types, 399
 text, 393

VIC chip, 406-407

Constrained menus, 32
 Control register settings, 20
 Count/bit-map pairs, 89-91, 96
 Count byte, 89-91
 C64Joystick, 166-167
 Current Mode, 127

D

Dabs, 195
 DBGETFILES, 225
 DblClickCount, 6
 DBOPVEC, 224
 DBTXTSTR, 220-221
 DBUSRICON, 224
 DB USR ROUT, 226
 DDec, 197
 Ddiv, 193
 Decimals, PutDecimal, 109
 DeleteFile, 266-267
 DeleteRecord, 322
 DeskTop, and icons, 26
 Diagonal lines, drawings of, 69, 71
 Dialog Boxes
 commands, 219, 222-223
 DBGETFILES, 225
 DBOPVEC, 224
 DBTXTSTR, 220-221
 DBUSRICON, 224
 DB USR ROUT, 226
 position byte, 219
 DoDlgBox, 218, 219, 231
 exiting from, 226
 icons, 219
 icon commands, 219-220
 LoadBox/GetFiles, 230
 openBox, 227-229
 position command byte, 218-219
 RecoverRectangle and, 226
 RecoverVector and, 226
 RstrFrmDialog, 218, 224, 226, 232
 SET DB POS, 218-219
 shadow box, 226
 structure of, 219
 Directory Block, 235, 237
 Directory Entry, 41, 43, 235, 238, 245
 Directory Header, 235, 236, 245
 format of, 49
 DisablSprite, 175
 Disk protection byte, 235
 Disk routines
 scope of, 245-246
 See also File system, disk routines.
 Disk variables, 424-425
 Dispatchers, 4
 Dispatch routines, menus, 32-33
 Display Buffering, graphics, 68-69
 DMult, 192
 Dnegate, 196
 DoDlgBox, 218, 219, 231
 DoIcons, 24, 27
 DoMenu, 36

- BBMult, 190
- BMult, 191
- Dabs, 195
- Ddiv, 193
- DMult, 192
- Dnegate, 196
- DSdiv, 194
- DShiftLeft, 188
- DShiftRight, 189
- GetRandom, 198
- Max char fault vector, 112
- MaximumMouseSpeed, 139, 141, 146
- Memory Map, 14-16, 410-413
 - file, 40
 - graphics, VICII chip definitions and equates, 413-414
 - pseudoregisters, 15
 - RAM, 15
 - zpage ares, 15
- Memory overlay techniques, 15
- Menus
 - constrained menus, 32
 - dispatch routines, 32-33
 - DoMenu, 36
 - DoPreviousMenu, 38
 - dynamic submenus, 33-34
 - File Quantum Test, 53-64
 - GotoFirstMenu, 39
 - horizontal menus, 29, 31, 33
 - menu actions, 31
 - menu selections, 30-34
 - ReDoMenu, 37
 - sample application, 40-43
 - submenus, 30
 - Change Departments, 31
 - Showcase Menu, 32
 - structures, 31
 - vertical menus, 29, 31, 33
- MinumMouseSpeed, 139, 146
- Mouse
 - position indicator on screen, 136
 - variables related to, 426-428
 - See also* Input driver.
- MouseAcceleration, 139, 141, 147
- MOUSE-BASE, 137
- MOUSE-BIT, 144-145
- MouseYposition, 136, 139, 143
- N
- Near-letter quality (NLQ) mode, dot-matrix printers, 327, 328
- NewDisk, 283
- NextRecord, 321
- Non-event code, 12
- Null terminating string, 104-105, 110
- NxtBlkAlloc, 293-294
- O
- OpenBox, 227-229
- OpenDisk, 246, 253
- OpenRecord File, 317, 318
- OtherPressVector, 5-6, 136
- OurGetString, 114, 116, 117-118, 127
- OurStringFault, 118
- OUT OF RECORDS, 316
- P
- Pen position, 99
- PLAINTEXT, 127
- Pointer to the current record, 316
- PointRecord, 321
- Points
 - dynamic submenus and, 130-131
 - PointSizeTable, 129, 130-131
- Position command byte, Dialog Boxes, 218-219
- Position escapes, text, 106-107
- PosSprite, 173
- PressFlag, 136, 139
- PreviousRecord, 321
- PRG format
 - File Header block, 45, 65-66
 - File Quantum Test, 53-64
 - PRGTOGEOS, 49-52
 - running program, 52
 - TestApplication, 47-48
- PRGTOGEOS, 49-52
- Primitive routines, 248-249
 - DoneWithIO, 307
 - EnterTurbo, 309
 - InitForIO, 306
 - PurgeTurbo, 308
 - ReadBlock, 310-311
 - WriteBloc, 312
- PrintASCII, 334, 341, 352
 - commodore driver, 373-374
- PRINTBASE, 332, 333
- PrintBuffer, 338, 341, 342, 344, 353
 - commodore driver, 369-370
- Print Driver Jump Table, 332, 346
- Print driver to disk stuff, commordore driver, 367
- Printer drivers
 - ASCII printing, 331, 333-334
 - CARDSWIDE/CARDSDEEP, 333
 - 8-bit printers, 344, 345-346
 - printer equates, 345
 - RAM storage/utilities, 346
 - resident jump table, 346
 - FormFeed, 360
 - GetDimensions, 331, 333, 336, 347
 - graphics printing, 330-331, 333
 - Greturn, 360
 - InitForPrint, 331-333, 335
 - InitPrinter, 357
 - PrintASCII, 334, 341, 352
 - PRINTBASE, 332, 333
 - PrintBuffer, 338, 341, 342, 344, 353
 - Print Driver Jump Table, 332, 346
 - PrintPrintBuffer, 355
 - Rotate, 361
 - sample driver, 342-344
 - SendBuff, 359
 - SetGraphics, 358
 - SetNLQ, 333, 351
 - StartASCII, 331-333, 340, 350
 - StartPrint, 331, 337, 340, 348-349
 - StopPrint, 339, 354
 - TestBuffer, 356
 - See also* Commodore driver.
- Printer equates, commodore driver, 365
- Printers, 325-329
 - character printers, 326
 - dot-matrix printers, 326-328
 - ASCII mode, 326-327, 328
 - GraphicsMode, 327
 - near-letter quality (NLQ) mode, 327, 328, 331
 - types by printheads, 327-328
 - parallel interface, 329
 - serial bus and, 328-329
- PrintPrintBuffer, 355
 - commodore driver, 375-376
- Process dispatcher, 177-178
- Process support
 - blocked process, 178
 - BlockProcess, 182
 - components of, 177-178
 - EnableProcess, 178, 186
 - FreezeProcess, 183
 - frozen process, 178
 - InitProcesses, 180
 - Process dispatcher, 177-178
 - RestartProcess, 181
 - runable process, 178
 - Sleep, 179, 184-185
 - UnblockProcess, 182
 - UnfreezeProcess, 183
- Programming steps, 13
- PromptOff, 122
- PromptOn, 116, 121
- Pseudoregisters, 10, 15
- PurgeTurbo, 308
- PutBlock, 274-275
- Putchar, 115, 123-14, 127
 - SmallPutChar, 124
- PutDecimal, 109
- PutDirHead, 282
- PutString, 104, 105-106, 108, 110
- R
- RAM
 - Memory Map, 15
 - start of, 423
 - storage/utilities, commodore driver, 366
- ReadBlock, 310-311
- ReadFile, 277
- ReadRecord, 324
- Records, VLIR files, 313
- RecoverAllMenus, 14
- RecoverLine, 14
- RecoverMenu, 14
- RecoverRectangle, 14, 87
 - Dialog Boxes, 226
- RecoverVector, 69

Dialog Boxes, 226
 Recovery routines, 14
 Rectangles, *See* Drawing filled regions.
 ReDoMenu, 37
 RenameFile, 268
 Resident jump table, commodore driver, 366
 Resident module, VLIR files, 313
 RestartProcess, 181
 Roll8bytesIn, commodore driver, 380
 Roll8bytesOut, commodore driver, 381
 RollaCard, commodore driver, 378
 ROM, 19
 Rotate, 361
 commodore driver, 385
 Routines
 equates for GEOS jump table, 431-436
 file, 40
 RstrFrmDialog, 218, 224, 226, 232
 Rts, 12
 Runnable process, 178

S

SaveFile, 264-265, 317
 SendBuff, 359
 SEQUENTIAL file, 233-234, 239
 Serial bus
 accessing, 249
 and printing, 328-329, 340
 Service routines, icons, 25-26
 SET DB POS, 218-219
 SetDevice, 246, 252
 SetGDirEntry, 298-299
 SetGEOSDisk, 255
 SetGraphics, 358
 commodore driver, 382
 SetNextFree, 295
 SetNLQ, 333, 351
 SetPattern, 80, 82
 Shadow box, Dialog Boxes, 226
 Showcase Menu, 32
 SineCosine, 168-169
 Sleep, 179-184-185
 SLOWMOUSE, 137, 139, 141-142, 159
 SmallPutChar, 124
 Sound interface device (SID), chip definitions and equates, 414-415
 Sprite support
 DisablSprite, 175
 DrawSprite, 172
 EnablSprite, 174
 PosSprite, 173
 Standard driver, 135
 StartASCII, 331-333, 340, 350
 commodore driver, 373
 StartPrint, 331, 337, 340, 348-349
 commodore driver, 368-369
 StopPrint, 339, 354
 commodore driver, 371
 StringFaultVector, 105, 115, 124
 String manipulation

GetString, 100-112, 104, 114, 118
 null terminating string, 104-105, 110
 PutString, 104, 105-106, 108, 110
 StringFaultVector, 105
 STRUCT MISMATCH, 317
 Style
 boldface, 127
 control of, 127-128
 current Mode, 127
 embedded style change characters, 105-106, 107
 italic, 127
 PLAINTEXT, 127
 Putchar and, 127
 Submenus, 30
 Change Departments, 31
 Showcase Menu, 32
 structures, 31
 Swap files, 238
 Swap modules, VLIR files, 313

T

TEMPORARY file type, 238
 Test Application, 47-48
 TestBuffer, 356
 commodore driver, 379
 TestPoint, 73
 Text
 embedded font change escape, 106-107
 max char fault vector, 112
 position escapes, 106-107
 PutDecimal, 109
 PutString, 104, 105-106, 108, 110
 string input, 104, 110-112
 Text prompt, 116, 120
 ToBasic, 212
 TopRollBuffer, commodore driver, 363, 364, 376

U

UnblockProcess, 182
 UnfreezeProcess, 183
 UNOPENED VLIR FILE, 316
 UnSetGraphics, commodore driver, 312
 UpdateMouse, 137, 143-145, 160
 Update MouseVels, 162-163
 ComputeMouseVels, 164
 UpdateMouseX, 165
 UpdateMouseY, 161
 UpdateRecordFile, 320
 UPLINE, 106
 User routines, 11
 UseSystemFonts, 130, 133
 Utility routines, 11

V

Variable Length Indexed Record, *See* VLIR files.
 Variables
 File System, 244-245

mouse variables

for applications, 152, 154
 for programming, 146-147
 Vertical lines, drawing of, 69-70, 75
 Vertical menus, 29, 31, 33
 VICII graphics, chip definitions and equates, 413-414
 VLIR files
 CloseRecordFile, 319
 creation of, 317
 DeleteRecord, 322
 error messages
 INVALID RECORD, 316
 OUT OF RECORDS, 316
 STRUCT MISMATCH, 317
 UNOPENED VLIR FILE, 316
 identification of, 315
 Index Table, 315
 "link list" concept of routines, 316
 NextRecord, 321
 OpenRecord File, 317, 318
 pointer to the current record, 316
 PointRecord, 321
 Previous Record, 321
 ReadRecord, 324
 records, 313
 resident module, 313
 structure of, 313-315
 swap modules, 313
 UpdateRecordFile, 320
 WriteRecord, 323

W

WarmStart Configuration,
 commodore driver, 386-390
 Word processing
 font changes, 106-107, 129-133
 style changes, 105-106, 107, 127
 See also Character level routines;
 Text.
 Wrap, text, 115
 WriteBlock, 312
 WriteRecord, 323

Z

Zero page, 417-419
 Zpage areas, 15

